University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

2016

ChilliSource Game Engine Particle System Study

Angela Gross
University of Montana, Missoula

Follow this and additional works at: https://scholarworks.umt.edu/etd

Part of the Other Computer Sciences Commons, Software Engineering Commons, Systems Architecture Commons, and the Theory and Algorithms Commons

Let us know how access to this document benefits you.

Recommended Citation

Gross, Angela, "ChilliSource Game Engine Particle System Study" (2016). *Graduate Student Theses, Dissertations, & Professional Papers*. 10813.
https://scholarworks.umt.edu/etd/10813

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.

CHILLISOURCE GAME ENGINE PARTICLE SYSTEM STUDY

By

Angela Nicole Gross

Bachelor of Science, The University of Montana, Missoula, MT, 2014

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

Autumn 2016

Approved by:

Scott Whittenburg Ph.D., Dean
Graduate School

Travis Wheeler Ph.D., Chair
Computer Science

Michael Cassens M.S.
Computer Science

Johnathan Bardsley Ph.D.
Mathematical Sciences

© COPYRIGHT

by

Angela Nicole Gross

2016

All Rights Reserved

Gross, Angela N., M.S., December 2016 Computer Science

ChilliSource Game Engine Particle System Study

Chairperson: Travis Wheeler

The majority of modern game engines utilize intricate objects called particle systems which are a collection of many particles that together represent an object without well-defined surfaces. This thesis discusses the results of studying and stressing particle systems within ChilliSource, an open-source game engine written in C++, with the goal of understanding a complex system and exploring possible optimizations that could be made to it. The studies performed were driven by metrics generated with custom profiling classes that kept track of things like the number of particles rendered, how long the engine spent rendering particles, or even how long a background thread that updated particles waited for a locked resource to release. These metrics supported experiments that revealed the inner workings of an elaborate system and aided in the creation and dissection of optimizations. The methods and results of these studies will aide anyone interested in reducing contention in large data structures either by using multiple mutexes, data structure "sharding", or hardware-based "lock free" implementations. They are also useful to any developer in need of profiling a complex system.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis advisor, Travis Wheeler, for providing me with the support, insight, and expertise necessary to complete my research. I would also like to thank a ChilliSource developer, Ian Copland, for his willingness to assist me with my study of the engine he created. Additionally, I am incredibly grateful to Andrea Johnson and Michael Breuer for their insightful comments that certainly improved my work. Without all of you, it would not have been possible to conduct my research.

TABLE OF CONTENTS

COPYRIGHT ii

ABSTRACT iii

ACKNOWLEDGMENTS iv

CODE LISTINGS ix

LIST OF FIGURES xii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 3

2.1 Particle Systems 3

2.1.1 Overview 3

2.1.2 Reeve's Particle Model 5

2.1.2.1 Life Cycle 5

2.1.2.2 Particle Attributes 5

2.1.2.3 Hierarchy 6

2.2 The ChilliSource Game Engine 7

2.2.1 Overview 7

2.2.2 Architecture 7

2.2.3 Life Cycle 9

2.3 ChilliSource Particle Effect Components 11

2.3.1 Overview 11

2.3.2 Architecture 12
2.3.2.1 Overview 12
2.3.2.2 Particle Effect 12
2.3.2.3 Particle Drawable 15
2.3.2.4 Particle Emitter 15
2.3.2.5 Particle Affector 16
2.3.2.6 Particle Array 17
2.3.2.7 Concurrent Particle Data 18
2.3.2.8 Other Important Attributes 18
2.3.3 Life Cycle 19
2.3.3.1 Overview 19
2.3.3.2 Initialization Phase 19
2.3.3.3 Update Phase 20
2.3.3.4 Render Phase 23
2.3.3.5 Clean Up Phase 24
2.3.4 Usage 24
CHAPTER 3 THE CHILLISOURCE AUTOMATED PONG GAME . . 27
3.1 Overview 27
3.2 Architecture 28
3.2.1 Overview 28
3.2.2 CSAPong Application 29
3.2.3 Application Systems 29
3.2.4 Game State 30
3.2.5 State Systems 30
3.3 Gameplay 31
3.3.1 Logistics 31

3.3.2 Examples 32
3.4 Initial Results 37
CHAPTER 4 CHILLISOURCE PARTICLE EFFECT COMPONENT OPTIMIZATION CASE STUDIES 40
4.1 Counting Contention 40
4.2 Using More Than One Mutex 42
4.2.1 Code 42
4.2.2 Results 46
4.2.3 Discussion 47
4.3 Using A Lock Free Data Structure 50
4.3.1 Code 51
4.3.2 Results 55
4.3.3 Discussion 58
4.4 Unexpected Results with Multiple Particle Effects 59
4.4.1 Using More Than One Mutex Results 60
4.4.2 Lock Free Results 62
4.4.3 Discussion 64
CHAPTER 5 CHILLISOURCE PARTICLE EFFECT COMPONENT OBSERVATION STUDIES 68
5.1 Overview 68
5.2 Copying Data with ConcurrentParticleData 68
5.2.1 Overview 68
5.2.2 Code 70
5.2.3 Results 73
5.2.4 Discussion 73
5.3 Multithreading and Task Scheduling 78

5.3.1 The ChilliSource Task Scheduler 78
5.3.2 Code 81
5.3.2.1 Small and Game Logic Task Type 82
5.3.2.2 No Task Scheduled (Not Threaded) 82
5.3.3 Results 83
5.3.4 Discussion 88
5.4 Particles Failing to Render 89
5.4.1 Render Failure 89
5.4.2 Emission Failure Source 90
5.4.3 Emission Failure Solution 91
CHAPTER 6 CONCLUSION 102
APPENDIX A INSTRUMENTATION APPENDIX 105
A.1 Overview 105
A.2 Metrics System 105
A.2.1 Metadata 105
A.2.2 Metrics 108
A.2.3 Particle Effect Definition 109
A.3 Timing System 110
A.3.1 Visual Studio 110
A.3.2 Shiny 112
A.3.3 Timing Application System 113
A.4 Output and its Evolution 114
BIBLIOGRAPHY 115

CODE LISTINGS

1 An example ChilliSource particle definition file for a generic particle effect. Note how the initial and changing-over-time (i.e. "affector") *Properties* can either be random values or constants. 13

2 The struct that is used to describe the properties of a particle within the `Particle Array` in *Particle Effect Component*. 17

3 The struct that is used to describe the properties of a particle within the `Particle Array` in *Concurrent Particle Data*. 18

4 The struct that is used to bundle all information needed by the background task performing the particle update. 20

5 Pseudocode describing how a *Particle Effect Component* is updated by scheduling a background task. 21

6 Pseudocode describing what `CommitParticleData` does when it copies its data to the *Concurrent Particle Data* instance. 22

7 Pseudocode showing how a *Particle Effect Component* uses its *Drawable* to render itself. 23

8 Creating a scene that adds a basic *Particle Effect Component* in ChilliSource. . . 24

9 The function to create an ambient light with the *Basic Entity Factory* state system. 25

10 Creating a basic particle effect component with ChilliSource. 26

11 Pseudocode illustrating the changes needed to utilize more than one mutex in the `CommitParticleData` function. Note that, instead of locking only one mutex, three different mutexes were locked at different times depending on the resource needed (i.e. the particles array, the new particle indices array, or the non-data structure array). 44

12 Pseudocode illustrating the changes needed to utilize more than one mutex in the `Draw` function. Note that, instead of locking only one mutex, two different mutexes were locked at different times depending on the resource needed (i.e. the particles array or the new particle indices array). 45

13 Pseudocode illustrating the behavior that a compare-and-swap operation. Although it is shown as a function here to demonstrate how it works, it is executed atomically via special hardware instructions. 52

14 The struct that is used to within the particle array in *`Concurrent Particle Data`* during the Lock Free case study. 53

15 Pseudocode illustrating the changes needed to make the `particles array` a lock free data structure in the `CommitParticleData` function. 53

16 Pseudocode illustrating the changes needed to make the `particles array` a lock free data structure in the `Draw` function. 54

17 Pseudocode describing the important changes needed to remove the `ConcurrentParticleData` instance from the *`Particle Effect Component`*. 71

18 Pseudocode describing the important changes needed to remove the `ConcurrentParticleData` instance from the `Particle Drawable`. 72

19 Pseudocode showing how ChilliSource's *`Task Scheduler`* schedules *small tasks*, *game logic tasks*, and *main thread tasks*. This provides context on how the scheduler schedules tasks and, most importantly, how the *game logic task* uses the `gameLogicCondition` member variable to notify the main thread when all game logic tasks are executed (see Code Listing 20). 80

20 Pseudocode showing how *game logic tasks* force the main thread to wait until they are finished before allowing the main thread to continue. See Code Listing 19 to see how the `gameLogicCondition` variable is used to notify the main thread that the game logic tasks are completed. 81

21 Pseudocode showing how `ParticleUpdateTask` can be scheduled as a *small task* and as a *game logic task*. 82

22 Pseudocode showing how `ParticleUpdateTask` was not scheduled and was simply run on the main thread. 82

23 An example invocation of the generate particles script. 109

24 Using the pseudocode from Code Listing 5, this shows how *`Shiny`* could be used to instrument `ParticleUpdateTask`. 113

25 Using the pseudocode from Code Listing 5, this shows how our custom *`Timing System`* could be used to instrument `ParticleUpdateTask`. 114

LIST OF FIGURES

Figure 2.1	Particle effect created by William T. Reeves	3
Figure 2.2	Cubes emitting particles	4
Figure 2.3	Basic structure of a ChilliSource application	8
Figure 2.4	An example of the Update life cycle event	11
Figure 2.5	Basic structure of a ChilliSource Particle Effect Component	13
Figure 3.1	A series of screenshots illustrating how the original pong game works.	28
Figure 3.2	The basic architecture of the automated pong game.	29
Figure 3.3	A series of screenshots showing the automated game with no particles.	33
Figure 3.4	A series of screenshots showing the automated game with a single, giant burst of particles.	34
Figure 3.5	A series of screenshots showing the automated game with multiple bursts of particle emissions.	35
Figure 3.6	A Visual Studio call tree sourced from the games based on the metadata in Figure 3.7.	38
Figure 3.7	Metadata of a series of games that demonstrated contention in `CommitParticleData`.	38
Figure 3.8	Timing output based on the metadata in Figure 3.7.	38
Figure 4.1	The results of "counting the contention" between the background and main threads in *Concurrent Particle Data*.	40

Figure 4.2 Illustrates how all of `Concurrent Particle Data`'s member variables are locked when their values are either being updated by a background thread in `CommitParticleData` or read from by the main thread in `Draw`. 43
Figure 4.3 Illustrates how the particles array and the rendering objects are given separate mutexes for the more than one mutex case study. 43
Figure 4.4 Illustrates the relationship between locking two mutexes between the drawing and updating threads over the lifetime of a particle effect. 44
Figure 4.5 The metadata used for the 8 series of automated games for the case studies. 46
Figure 4.6 More Than One Mutex Case Study: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. 48
Figure 4.7 More Than One Mutex Case Study: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. 48
Figure 4.8 More Than One Mutex Case Study: A graph showing the number of section calls for various sections of code for a single particle effect. 49
Figure 4.9 More Than One Mutex Case Study: A graph showing the time spent in various sections of code for a single particle effect. 49
Figure 4.10 Shows the particle array with a mutex for every particle element (left) and with an atomic variable for every particle element (right). 50
Figure 4.11 Lock Free Case Study: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. 56
Figure 4.12 Lock Free Case Study: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. 56
Figure 4.13 Lock Free Case Study: A graph showing the number of section calls for various sections of code for a single particle effect. 57

Figure 4.14 Lock Free Case Study: A graph showing the time spent in various sections of code for a single particle effect. 57

Figure 4.15 The metadata used for the 8 series of automated games for the unexpected results from the case studies. 59

Figure 4.16 More Than One Mutex Case Study: A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. 60

Figure 4.17 More Than One Mutex Case Study: A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. 60

Figure 4.18 More Than One Mutex Case Study: A graph showing the number of section calls for various sections of code for ten particle effects. . 61

Figure 4.19 More Than One Mutex Case Study: A graph showing the time spent in various sections of code for ten particle effects. 61

Figure 4.20 Lock Free Case Study: A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. 62

Figure 4.21 Lock Free Case Study: A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. 62

Figure 4.22 Lock Free Case Study: A graph showing the number of section calls for various sections of code for ten particle effects. 63

Figure 4.23 Lock Free Case Study: A graph showing the time spent in various sections of code for ten particle effects. 63

Figure 4.24 A summary of the timed sections for the unexpected results for the Lock Free and > 1 Mutex case studies for a single particle effect. . 65

Figure 4.25 A summary of the timed sections for the unexpected results for the Lock Free and > 1 Mutex case studies for ten particle effects. . . . 65

Figure 4.26 The lifetime of a single particle effect in relation to its scheduled background thread and the main thread. 66

Figure 4.27 The lifetime of a many particle effects in relation to their scheduled background threads and the main thread. 66

Figure 5.1 Demonstrates how the updating and rendering threads use the `ConcurrentParticleData` object to communicate information about the particle effect. 69

Figure 5.2 Demonstrates how, during this observation study, updating and rendering threads worked directly with the particle effect data. 69

Figure 5.3 Using Copies Observation: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. . . . 74

Figure 5.4 Using Copies Observation: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. . . . 74

Figure 5.5 Using Copies Observation: A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. 75

Figure 5.6 Using Copies Observation: A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. 75

Figure 5.7 Using Copies Observation: A graph showing the number of section calls for various sections of code for a single particle effect. 76

Figure 5.8 Using Copies Observation: A graph showing the time spent in various sections of code for a single particle effect. 76

Figure 5.9 Using Copies Observation: A graph showing the number of section calls for various sections of code for ten particle effects. 77

Figure 5.10 Using Copies Observation: A graph showing the time spent in various sections of code for ten particle effects. 77

Figure 5.11 Illustrates the lifetime of *small tasks*, *game logic tasks*, and *main thread tasks* in relation to the main thread. 79

Figure 5.12 Illustrates the relationship between the updating and rendering processes when the updating thread is defined as a *small task*, *game logic task*, and when the updating process is not parallelized (i.e. not threaded). 79
Figure 5.13 Task Scheduling Observation: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. . . . 84
Figure 5.14 Task Scheduling Observation: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. . 84
Figure 5.15 Task Scheduling Observation: A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. 85
Figure 5.16 Task Scheduling Observation: A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. . . . 85
Figure 5.17 Task Scheduling Observation: A graph showing the number of section calls for various sections of code for a single particle effect. . . 86
Figure 5.18 Task Scheduling Observation: A graph showing the time spent in various sections of code for a single particle effect. 86
Figure 5.19 Task Scheduling Observation: A graph showing the number of section calls for various sections of code for ten particle effects. 87
Figure 5.20 Task Scheduling Observation: A graph showing the time spent in various sections of code for ten particle effects. 87
Figure 5.21 Demonstrates the difference between looping and non-looping particle effects that both have a duration of 3 seconds. 90
Figure 5.22 Emission Failure Observation: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. . . . 92
Figure 5.23 Emission Failure Observation: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. . 92

Figure 5.24 Emission Failure Observation: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. . . . 93
Figure 5.25 Emission Failure Observation: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. . 93
Figure 5.26 Emission Failure Observation: A graph showing the number of section calls for various sections of code for a single particle effect. . . 94
Figure 5.27 Emission Failure Observation: A graph showing the time spent in various sections of code for a single particle effect. 94
Figure 5.28 Emission Failure Observation: A graph showing the number of section calls for various sections of code for ten particle effects. 95
Figure 5.29 Emission Failure Observation: A graph showing the time spent in various sections of code for ten particle effects. 95
Figure 5.30 Emission Failure Solution Observation: A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. 98
Figure 5.31 Emission Failure Solution Observation: A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. 98
Figure 5.32 Emission Failure Solution Observation: A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. 99
Figure 5.33 Emission Failure Solution Observation: A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. 99
Figure 5.34 Emission Failure Observation: A graph showing the number of section calls for various sections of code for a single particle effect. . . 100
Figure 5.35 Emission Failure Observation: A graph showing the time spent in various sections of code for a single particle effect. 100
Figure 5.36 Emission Failure Observation: A graph showing the number of section calls for various sections of code for ten particle effects. 101

Figure 5.37 Emission Failure Observation: A graph showing the time spent in various sections of code for ten particle effects. 101

Figure A.1 An example of metadata output by the *Metrics System*. 106

Figure A.2 An example of the changing particle values during games in CSAPong. 106

Figure A.3 An example of metrics output by the *Metrics System* based on the metadata from Figure A.1. 108

Figure A.4 Shows the parameters that the generate_particles.py script can use. 110

Figure A.5 Shows the particle files that were generated by an invocation like in Code Listing 23. 110

Figure A.6 Shows an example of Visual Studio's generated call tree for CPU instrumentation when ran using the metadata from Figure A.1. . . . 111

Figure A.7 Shows the same call tree from Figure A.6, but sorted and filtered. . 111

Figure A.8 An example of timing output by the *Timing System* with just the ParticleUpdateTask and the *Particle Iteration* times. 114

CHAPTER 1 INTRODUCTION

Game engines can broadly be defined as frameworks that developers can use to create games. Many engines include 2D and 3D rendering, sound effects, responsive user interfaces, lighting, and cross-platform support. Some allow the developer to easily create games through an editor via dragging and dropping, while others only offer a repository of code to include as a library. A number of game engines also support the creation of particle systems, or a collection of many particles that together represent an object without well-defined surfaces. Some examples of particle systems are fireworks, snow, oceans, fire, and smoke. ChilliSource, an open-source game engine written in C++, supports the creation and usage of particle systems. Chapter 2 provides a more in-depth discussion on what particle systems are, the structure of ChilliSource, and how particle systems fit into that structure. This is discussed at great length in this thesis because the particle systems within ChilliSource were studied and stressed with the goal of understanding a complex system and exploring possible optimizations that could be made to it. An incomplete understanding of particle systems, ChilliSource, and ChilliSource's particle systems may have resulted in poorly constructed optimizations, and that would have negatively impacted my studies.

In order to aide in stressing and instrumenting the ChilliSource engine, I created an automated pong game. This automated game can run a series of "games" where the ball bounces around the arena without any user intervention. The various kinds of games differ in the particle systems that are attached to the ball. For example, one game could have an attached particle system that only emitted 10 particles every 2 seconds, while another

game could have an attached particle system that emitted 1000 particles every 5 seconds. When any given automated game reaches completion, it saves information (metrics) about the games that ran. Some metrics include the number of particles that emitted or how long the engine spent drawing particles. The metrics generated by the automated game drove all of the studies that were performed. For example, metrics relating to how long certain areas of code ran was primarily used in studies that focused on minimizing contention in a large data structure. Chapter 3 explains how exactly these games were ran and directly relates the produced metrics to the automated game, and this is essential to understand because it provides a context to my studies and their results.

Chapters 4 and 5 cover the methods and results of the two kinds of studies that I performed: optimization case studies and observation studies.

The optimization case studies chapter focuses on the methods and results of optimizing a background task that updates information about each individual particle within a particle system (e.g. current velocity, position, size, color, etc.). I sought to reduce contention in a large data structure either by using multiple mutexes, data structure ”sharding”, or hardware-based ”lock free” implementations.

The observation studies chapter, on the other hand, explores other aspects of ChilliSource's particle system. The covered topics in these observation studies are the benefits of encapsulating thread synchronization within a single object, the relationship between ChilliSource's task scheduler and its particle systems, and an interesting scenario in which particles did not visually appear.

CHAPTER 2 BACKGROUND

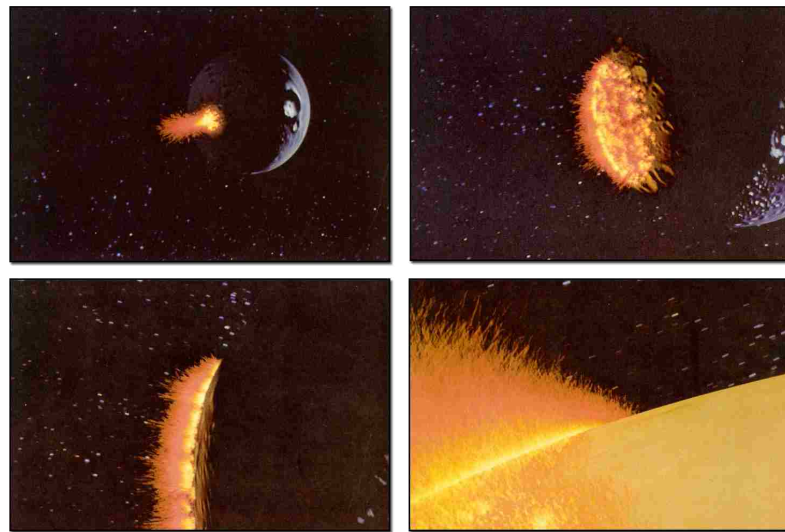

Figure 2.1: The particle effect created by William T. Reeves from the film Star Trek II: The Wrath of Khan

2.1 Particle Systems

2.1.1 Overview

Visual representation of "fuzzy objects", or objects without well-defined surfaces, in computer graphics is generally considered a difficult task due to the non-deterministic changes in the fuzzy object's shape and form. When working on the film *Star Trek II: The Wrath of Khan*, a researcher at Lucasfilm Ltd., William T. Reeves, developed what he called a ***particle system***, which is "... a collection of many minute particles that together represent a fuzzy object. Over a period of time, particles are generated into a system, move and change

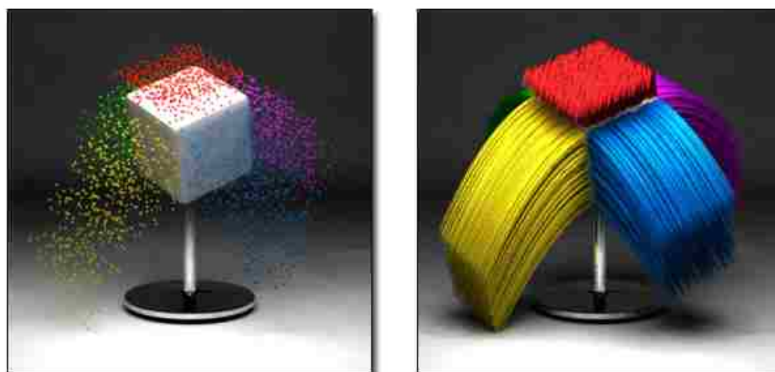

Figure 2.2: Left: A cube emitting a stream of 1000 animated particles from 5 of its 6 faces. Right: A cube emitting a stream of 1000 static particles, or strands, from 5 of its 6 faces

from within the system, and die from the system." [1] Although the concept of modeling these objects as a collection of particles was not novel, Reeves' basic framework for particle systems is still widely used today to create complex visual effects in video games and films.

Fuzzy objects like sparks, clouds, stars, water, and crowds of people can be easily translated and modeled as a particle system; particle systems can also be used to model objects that contain numerous strands like hair and grass. These objects can be represented by rendering every strand as the entire lifetime of a particle in the system and is then treated as a single strand. The particles in a system that have a distributed lifetime (e.g. sparks) are thought of as ***animated*** and particles that are rendered all at once (e.g. hair) are thought of as ***static***. For the duration of our own study, we have exclusively focused on animated particles.

A solid understanding of how particle systems work and how ChilliSource operates was essential in my development of optimizations for ChilliSource's particle systems. Knowing the attributes and events that are necessary to maintain a particle system guided what kinds of changes that I could and could not make. Additionally, knowledge of ChilliSource's life cycle and architecture provided an essential context on how ChilliSource facilitates communication between its particle systems, the user, and its renderer (i.e. a system that

draws objects to the screen). This knowledge fueled the optimizations and the studies that were performed in this thesis.

2.1.2 Reeve's Particle Model

2.1.2.1 Life Cycle

A particle system has 4 main events that defines its life cycle:

1. Emission (or Generation, according to Reeves)
2. Update
3. Render
4. Death (or Extinction, according to Reeves)

The **Emission** event is the initial process in which particles are generated with their initial values. Often a "generation shape"[1] is used during the emission phase that defines the two or three dimensional space that the particles can originate, or emit, from. Emission can be increased or decreased in frequency using the ***emission rate***, and the number of particles that enter the particle system each emission can also be changed using the ***particles per emission*** variable. After emission, particles in the particle system undergo the **Update** and **Render** events. These events are repeated until the particle dies, or undergoes the **Extinction** phase. The **Update** phase involves iterating over all particles throughout the system and updating their attributes (e.g. position, velocity, etc.). The **Render** phase, on the other hand, is the phase in which all of the particles are actually drawn to the screen.

2.1.2.2 Particle Attributes

As mentioned above, each individual particle in a particle system will have attributes that change over time. Although every particle system is different, the following is what Reeves considered to be the core attributes of a particle:

- Position

- Velocity
- Size
- Color
- Transparency
- Shape
- Lifetime

A particle has a position in two or three dimensional space, and its initial position is determined by the emission process. Other attributes that may also change over time include its speed, direction, size, color, and transparency. Its lifetime, however, is predetermined at emission, and it is the attribute that dictates how long the particle "lives", or remains on the screen. In order to decrease uniformity and increase realism, all of these values can vary across the particles in the system. For example, the particles in a given particle system could have a varying lifetime of 5 to 10 seconds so that the particles that shared an emission would not all die simultaneously.

2.1.2.3 Hierarchy

It is often convenient to design a particle system such that it is made up of a hierarchy of child particle systems. This hierarchy "can be used to exert global control on a complicated fuzzy object that is composed of many particle systems." [1] In other words, a fuzzy object could easily be transformed in overall shape, size, color, etc. by descending into and transforming its child particle systems. It also allows the grouping together of particle systems that have different types of particles. For example, a campfire fuzzy object would need smoke, sparks, and the flame itself. All of these components would have distinct particle definitions and would be separate particle systems. A particle system hierarchy, in this case, would allow the smoke, sparks, and flame particle systems to be bundled together as a single fuzzy object.

2.2 The ChilliSource Game Engine

2.2.1 Overview

Chosen for its open source and well-thought out architecture, ChilliSource is a free, open source game engine created by ChilliWorks which is a group that is part of United Kingdom's Tag Games. It allows developers to create 2D and 3D content for iOS, Android, Kindle, and Windows operating systems in C++ 11. Some of its key features include networking, shader support, responsive (i.e. adaptive to different resolutions) graphical user interfaces, modular and extensible lighting and shadows, skinned animation, Facebook connect integration, consistent API, and automatic builds of iOS/Android/Kindle applications. [2]

2.2.2 Architecture

The structure of a ChilliSource application is primarily composition based in that it relies on a hierarchy of modules to define its functionality. The modules comprising the application are *States*, *Systems*, *Entities* and *Components*. Figure 2.3 demonstrates the architecture of a ChilliSource application using these basic modules.

The *Application* instance initializes and manages *Application Systems*, which are modules that describe functionality used across the entire application. A non-exhaustive list of some application systems include the *Task Manager*, *Texture Provider*, *Render System*, and the *State Manager*. The *State Manager*, as shown in Figure 2.3, manages a stack of *States* and ensures that the state on top of the stack is active.

States can be thought of as different parts of the application such as a main menu, inventory, settings menu, pause screen, and the game scene itself. *States*, like the *Application*, initialize and manage *State Systems*. Similar to *Application Systems*, *State Systems* describe functionality to be used by one state and only exist during that state's lifetime. The base *State* creates and manages the *Canvas*, *Gesture*, and user-defined state systems. The *Canvas* renders the user interface, the *Gesture* state system helps receive input events like pinching or tapping, and user-defined state systems are cus-

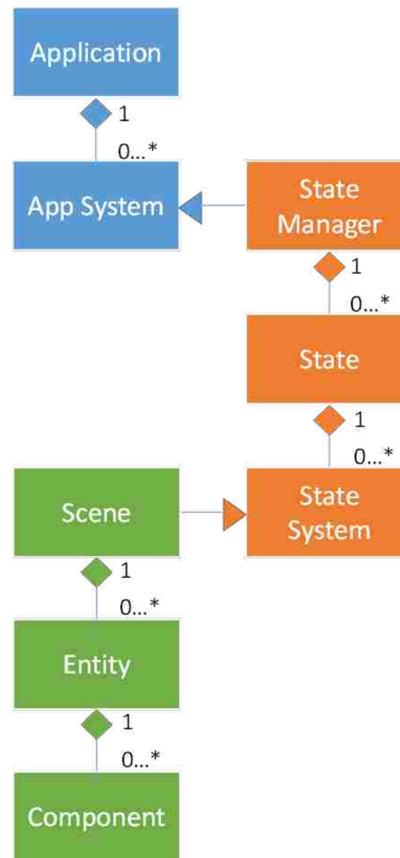

Figure 2.3: The basic structure of a ChilliSource application

tom modules built by the developer to describe functionality for a state. An important state system, the `Scene`, manages `Entities` that form that scene.

`Entities` are essentially objects within the game that are defined by both (a) a transform describing their rotation, size, and position and (b) `Components`, which are modules that describe the entity's functionality. Some examples of `Components` are AI Controllers, Health, Damage, and Physics. In addition to transforms and `Components`, `Entities` can also contain a hierarchy of entities in which, much like in particle systems, their transforms and life cycle events are linked by the hierarchy.

ChilliSource's architecture favors composition over inheritance. For example, if we wanted to create enemy and player `Entities`, we may be tempted to create a base Human `Entity`

class that had health related methods and attributes. However, in order to truly adhere to the spirit of ChilliSource's design, we would create a *Component* with a name like *HealthComponent* that could be attached to any single *Entity*. In this way, ChilliSource lends itself to a great deal of code reusability and flexibility.

2.2.3 Life Cycle

The majority of game engines follow a life cycle very similar to the one for particle systems that Reeves developed, which is:

1. Initialize
2. Update
3. Draw
4. Stop

The **Initialize** step usually involves loading textures, sounds, and anything else that the engine needs to do before starting the main game loop. The **Update** step is the heart of game, and is usually driven by some kind of state machine. Some examples of typical states include Menu, Game, Settings, and Exit. The **Draw** step is also part of the main game loop and, just as it sounds, this is where everything is drawn to the screen. Typically, there is some kind of renderer that handles all of the low-level calls to the graphics card (e.g. OpenGL, DirectX, etc.). Optionally, there is the **Stop** step in which the game has been stopped and all assets and resources will be released. This may or may not be included as a separate state in the **Update** phase, but it is good to logically separate the two events.

ChilliSource, as one may expect, adheres to the basic **Initialize**, **Update**, **Draw**, and **Stop** phases. However, there are a great deal of other events that are perpetuated from the *Application* instance. The following is a list of the main life cycle events that all modules receive:

1. Init

- This initializes the application and begins the **Update** event.

2. Update
   - This updates the application, its systems, and the state manager's states.
3. Render
   - This renders objects and user interface elements to the screen and is the main draw loop of the application.
4. Suspend
   - This notifies active state(s) to pause and it suspends all systems in reverse order.
5. Resume
   - This resumes the application from a suspended state.
6. Background
   - This occurs when the application is not in view any longer.
7. Foreground
   - This occurs when the application is in view again.
8. Destroy
   - This cleans up the application and releases all systems in reverse order.

All of these events originate from the *Application* instance. The flow of events looks something like *Application* → *Application Systems* (*State Manager*) → *States* → *State Systems* (*Scene*) → *Entities* → *Components*. Figure 2.4 shows what this flow looks like for the **Update** life cycle event using pseudocode. Note, however, that each event means something slightly different depending on which module it is called in (e.g. **Update** in the *Application* means something different than the **Update** in a *State*). Additionally, there are a number of other events not discussed here that only have meaning for specific modules (e.g. `OnAddedToEntity` for the *Component* module). See [3] for more information.

The game developer is not limited to the main events that are automatically pushed by the application. Custom events can also be created that are subscribed to by entities,

components, and systems. When an object subscribes to a custom event, this object utilizes a `Connection` to listen for the event it subscribed to. An `Event` can open, close, and notify the `Connections` other objects have made to it. Some examples of custom events are entity collisions, player death, cut scene trigger, game lost, game won, and so on.

2.3 ChilliSource Particle Effect Components

2.3.1 Overview

A special type of component in ChilliSource is the `Particle Effect Component`. It inherits from the `Render Component` class [1], which is, just as it sounds, a component that is rendered to the screen. `Render Components` have methods for visibility, culling, casting shadows, and usage of both its transformation matrix and sort predicates. They also have virtual methods that child classes can define for materials, transparency, rendering, and bounding boxes. For the purpose of this study, however, a discussion of the `Render Component`'s details is outside of scope. It is only necessary to know that the `Particle Effect Component` is not only updated by its parent entity but also is rendered by the

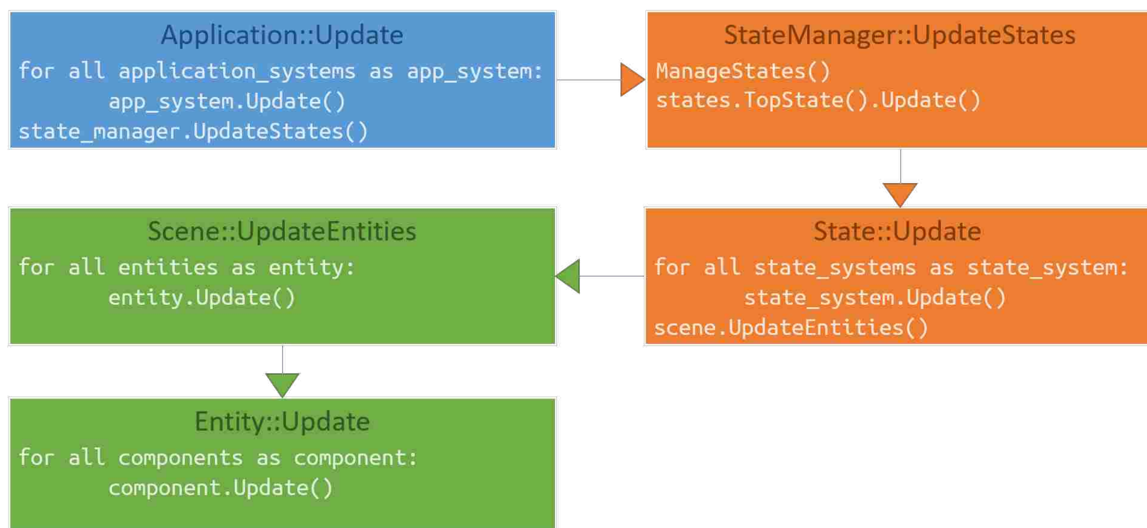

Figure 2.4: An example of the **Update** life cycle event being perpetuated from the `Application` all the way down to a `Component`

Application's renderer.

2.3.2 Architecture

2.3.2.1 Overview

The structure of a *Particle Effect Component* in ChilliSource is fairly complex. Its composition includes a number of objects, primitives, and data structures, but we can boil it down to 6 parts:

- A *Particle Effect* object
- A *Particle Drawable* object
- A *Particle Emitter* object
- A list of *Particle Affectors* (optional)
- A *Concurrent Particle Data* object
- An array of *Particle* structs

2.3.2.2 Particle Effect

The *Particle Effect* object contains all of the *Properties* and *Definitions* for the effect. A *Property* defines a value that may be more complex than just a single value. For example, a *Property* could be defined as a random value selected within a certain range, a value that changes over the lifetime of the particle effect, or simply a constant primitive value. *Definitions*, on the other hand, describe a number of properties that should be used to create an object. The *Properties* and *Definitions* that the *Particle Effect* contains include:

- The particle effect's duration
- The total maximum number of particles used in the effect

[1] It's important to note here that the version of ChilliSource that we are using is 1.6.0 and, at the time of writing this paper, it is currently at version 2.0.0. In version 2.0.0, the rendering system is completely overhauled and *Particle Effect Components* are *Volume Components* and not *Render Components*. This new system was not studied and will not be discussed in depth here.

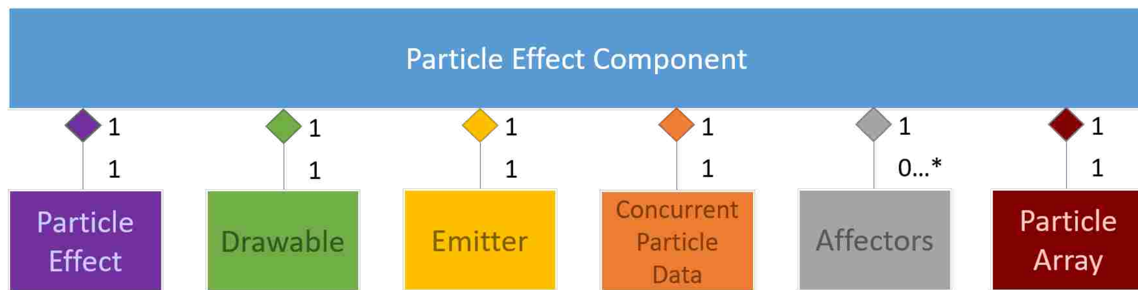

Figure 2.5: The basic structure of a ChilliSource Particle Effect Component

- The space in which the particle effect is simulated (either "local" or "world")
- The lifetime property of a new particle generated in the effect
- The initial scale property of a new particle generated in the effect
- The initial rotation property of a new particle generated in the effect
- The initial color property of a new particle generated in the effect
- The initial speed property of a new particle generated in the effect
- The initial angular velocity property of a new particle generated in the effect
- The particle effect's drawable definition
- The particle effect's emitter definition
- The particle effect's affector definitions

These *`Properties`* and *`Definitions`* are used to create instances and values that will be used by the *`Particle Effect Component`*. Thus, the contents of the *`Particle Effect`* instance completely determines the behavior and appearance of the *`Particle Effect Component`*. In order to create these intricate *`Particle Effects`*, a JSON particle definition file like the one shown in Code Listing 1 must be utilized. Given the length and verbosity of this file, it is easy to see that defining a particle effect by hand in C++ would be undesirable.

Code Listing 1: An example ChilliSource particle definition file for a generic particle effect. Note how the initial and changing-over-time (i.e. "affector") *`Properties`* can either be random values or constants.

```
{
  "Duration": "1.0",
  "MaxParticles": "10000",
  "SimulationSpace": "World",
  "LifetimeProperty": {
    "Type": "RandomConstant",
    "LowerValue": "0.8",
    "UpperValue": "1.2"
  },
  "InitialScaleProperty": {
    "Type": "RandomConstant",
    "LowerValue": "0.5 0.5",
    "UpperValue": "1.2 1.2"
  },
  "InitialRotationProperty": "0.0",
  "InitialColourProperty": "0.0 0.0 0.0 0.0",
  "InitialSpeedProperty": "0.5",
  "InitialAngularVelocityProperty": "0.0",
  "Drawable": {
    "Type": "StaticBillboard",
    "MaterialPath": "Particle/Particle.csmaterial",
    "AtlasPath": "TextureAtlases/Particle/Particle.csatlas",
    "ImageIds": "Generic",
    "Size": "1.0 1.0",
    "SizePolicy": "FitMaintainingAspect"
  },
  "Emitter": {
    "Type": "Cone",
    "EmissionMode": "Stream",
    "EmitFromType": "Inside",
    "EmitDirectionType": "AwayFromBase",
    "EmissionRateProperty": "100.0f",
    "ParticlesPerEmissionProperty": "10000",
    "EmissionChanceProperty": "1.0",
    "RadiusProperty": "0.5",
    "AngleProperty": "0.8"
  }
  "Affectors": [
    {
      "Type": "ColourOverLifetime",
      "TargetColourProperty": "0.0 0.0 0.0 0.0",
      "IntermediateColours": [
        {
          "ColourProperty": {
            "Type": "RandomConstant",
            "LowerValue": "0.9 0.4 0.12 0.0",
            "UpperValue": "1.0 0.25 0.0 0.5"
          },
          "TimeProperty": "0.15"
        },
        {
          "ColourProperty": {
            "Type": "RandomConstant",
            "LowerValue": "0.9 0.4 0.12 0.0",
            "UpperValue": "1.0 0.25 0.0 0.5"
```

```
56            },
57            "TimeProperty": "0.75"
58          }
59        ]
60      },
61      {
62        "Type": "ScaleOverLifetime",
63        "ScaleProperty": {
64          "Type": "RandomConstant",
65          "LowerValue": "0.75 1.5",
66          "UpperValue": "1.0 2.0"
67        }
68      }
69    ]
70  }
```

2.3.2.3 Particle Drawable

The *Drawable* object is responsible for rendering all of the particles that belong to its *Particle Effect Component*. A *Static Billboard Particle Drawable*[2], which is a *Drawable* that renders each and every individual particle as camera facing sprites, is used to accomplish this. The *Drawable* always renders particles on the main game thread.

2.3.2.4 Particle Emitter

The *Emitter* object is used to spawn new particles during **Emission**. The *Emitter* gets to decide the direction, location, and moment that a new particle emits. Unlike the rendering that a *Drawable* does, the *Emitter* emits particles as a background task. There are 5 different kinds of *Emitters*, and their differences lie in the ”generation”, or emission, shape:

- *Point Particle Emitter*
  - This emitter spawns particles at the parent entity's position with a random direction.

[2] As of version 1.6.0, the only *Drawable* type available to us is the *Static Billboard*, but the current architecture suggests that there could be different types of *Drawables* in future versions.

- *Circle Particle Emitter*
  - This emitter spawns particles either within a circle or on the circle's perimeter. It can be set to emit particles with a random direction or a direction away from the center of the circle.
- *Sphere Particle Emitter*
  - Similar to the *Circle Particle Emitter*, this emitter spawns particles either within a sphere or on the sphere's surface. It can be set to emit particles with a random direction or a direction away from the center of the sphere.
- *Cone Particle Emitter*
  - This emitter spawns particles either within a cone or on the cone's surface. It can be set to emit particles with a random direction or a direction away from the base of the cone.
- *Cone 2D Particle Emitter*
  - This emitter spawns particles either within a 2D cone, on the 2D cone's edges, or at the 2D cone's base. It can be set to emit particles with a random direction or a direction away from the base of the 2D cone.

2.3.2.5 Particle Affector

A *Particle Affector* object applies an effect to all particles that change over time. A *Particle Effect Component* can utilize zero or more *Affectors*, and, like the *Emitter*, *Affectors* update particles in a background thread. There are 4 different kinds of *Affectors* that can be applied:

- *Scale Over Lifetime Particle Affector*
  - This affector changes the size of the particles over their lifetime by using a calculated scale factor. This factor will be used to help interpolate the particle's size across updates.
- *Color Over Lifetime Particle Affector*

  - This affector changes the color of particles over their lifetime by using its initial and target colors to generate interpolated colors across updates.
- *Acceleration Lifetime Particle Affector*
  - This affector changes the speed of particles over their lifetime.
- *Angular Acceleration Lifetime Particle Affector*
  - This affector changes the angular velocity of particles over their lifetime.

2.3.2.6 Particle Array

The `Particle Array` is a data structure that holds all of the particles that comprise the *Particle Effect Component*, and it is used by the *Drawable*, *Emitter*, *Affector*, and *Concurrent Particle Data* instances. Each particle is represented by a *Particle* struct, as shown in Code Listing 2. We can observe that the information provided by this struct mirrors the particle attributes that Reeves used in his particle systems.

Code Listing 2: The struct that is used to describe the properties of a particle within the `Particle Array` in *Particle Effect Component*.

```
1  struct Particle final
2  {
3      bool m_isActive = false;
4      f32 m_lifetime = 0.0f;
5      f32 m_energy = 0.0f;
6      Vector3 m_position;
7      Vector2 m_scale = Vector2::k_one;
8      f32 m_rotation = 0.0f;
9      Colour m_colour = Colour::k_white;
10     Vector3 m_velocity;
11     f32 m_angularVelocity = 0.0f;
12 };
```

2.3.2.7 Concurrent Particle Data

The *Concurrent Particle Data* object is a container for data that is shared by the background and main threads. *Concurrent Particle Data* is written to by the same background thread that emits and affects particles, and it is read by the *Particle Effect Component*'s *Drawable* in the main thread. Essentially, it contains copies of important attributes that these threads share. This includes an array of particles, indices of particles that were just emitted, and the bounds of the particle effect. Thus, all thread synchronization and locking occurs in *Concurrent Particle Data* and not in the *Particle Effect Component* itself.

It is important to note that *Concurrent Particle Data*'s array of particles uses a different struct than the one used in *Particle Effect Component*. As we can see in Code Listing 3, the struct contains a subset of the information that is provided by *Particle Effect Component*'s *Particle* struct. This is intentional, as information that will be used by the *Drawable* later should only be copied.

Code Listing 3: The struct that is used to describe the properties of a particle within the `Particle Array` in *Concurrent Particle Data*.

```
struct Particle final
{
    bool m_isActive = false;
    Vector3 m_position;
    Vector2 m_scale = Vector2::k_zero;
    f32 m_rotation = 0.0f;
    Colour m_colour = Colour::k_white;
};
```

2.3.2.8 Other Important Attributes

There are a number of other important attributes that are not the 6 parts discussed above. These include:

- Bounding box objects

- Playback enumerators and floats
- Custom Events

The bounding box objects are containers that describe the effect's shape and are used to aide in the rendering of the effect. The playback enumerators- *Playback Type* and *Playback State*- help describe whether or not the particle effect loops and the current life cycle phase of the particle effect, respectively. The playback floats keep track of how long the particle effect has been playing. The custom events that the effect uses are fired off when the particle effect fully completes (i.e. no more particles are left alive) and when a particle effect finishes emitting.

2.3.3 Life Cycle

2.3.3.1 Overview

The life cycle of a *Particle Effect Component* is very similar to the one developed by Reeves, and it can be described as 4 main phases:

1. Initialization
2. Update
3. Render
4. Clean Up

2.3.3.2 Initialization Phase

During the **Initialization** phase, the *Particle Effect Component* creates the *Drawable*, *Emitter*, *Concurrent Particle Data*, and, if specified by the *Particle Effect* instance, its *Affectors*. It also creates the `Particle Array` and initializes the *Bounding Shapes*.

2.3.3.3 Update Phase

The **Update** phase uses 4 states to characterize the current playback state of the particle effect, which are `Starting`, `Playing`, `Stopping`, and `Not Playing`. During the `Starting` state, particles are initialized to be inactive before it updates the particle effect for the first time and enters the `Playing` state. The `Playing` state is when particles are rendered and updated. During the `Playing` state, the particle effect may enter a `Stopping` state if the playback timer exceeds the effect's duration. However, it may return to a `Playing` state if active particles still remain within the `Particle Array`. When the particle effect's duration time has elapsed and no active particles remain, the particle effect finally enters the `Not Playing` state and the particle effect no longer does anything. The bulk of a particle effect's time is spent during the `Playing` state, and that is the state of the **Update** phase that will be focused on throughout this thesis.

Code Listing 4: The struct that is used to bundle all information needed by the background task performing the particle update.

```
struct ParticleUpdateDesc final
{
    ParticleEffectCSPtr m_particleEffect;
    ParticleEmitterSPtr m_particleEmitter;
    std::vector<ParticleAffectorSPtr> m_particleAffectors;
    std::shared_ptr<dynamic_array<Particle>> m_particleArray;
    ConcurrentParticleDataSPtr m_concurrentParticleData;
    f32 m_playbackTime = 0.0f;
    f32 m_deltaTime = 0.0f;
    Vector3 m_entityPosition;
    Vector3 m_entityScale;
    Quaternion m_entityOrientation;
    bool m_interpolateEmission = false;
};
```

When the particle effect is updating, it prepares a struct containing pertinent information to hand over to a background task. This struct contains pointers to the *`Particle Effect`*, *`Emitter`*, *`Affector(s)`*, `Particle Array`, and *`Concurrent Particle Data`* variables. Additionally, it holds some other miscellaneous information such as the

effect's parent entity's position, scale, and orientation. The struct in its entirety is shown in Code Listing 4. After it prepares the struct, it uses the *Application Task Scheduler* application system to schedule a background task. This process is shown using pseudocode in lines 1 - 11 in Code Listing 5.

As mentioned before in Section 2.3.2, the `Particle Update Task` updates particles from the `Particle Array`, applies *Affectors* to the particles, emits particles using the *Emitter*, calculates its new *Bounding Shapes*, and updates *Concurrent Particle Data* with the new information. Lines 13 - 41 demonstrate these actions in Code Listing 5.

Code Listing 5: Pseudocode describing how a *Particle Effect Component* is updated by scheduling a background task.

```
1  ParticleEffectComponent::OnUpdate()
2  {
3    // Get a struct containing copies (and pointers) to pass to the task
4    copiedAtts = CopyUpdateAttributes();
5
6    // Schedule a task to update the particle in the background
7    ApplicationTaskScheduler.ScheduleTask
8    ({
9      ParticleUpdateTask(copiedAtts);
10   });
11 }
12
13 ParticleUpdateTask(copiedAtts)
14 {
15   // Update particles if they are active
16   for(particle in copiedAtts.particleArray)
17   {
18     if(particle.isActive)
19       particle.UpdateValues();
20   }
21
22   // Apply affectors
23   for(affector in copiedAtts.particleAffectors)
24   {
25     affector.AffectParticles();
26   }
27
28   // Try to emit
29   newParticleIndicesArray = copiedAtts.particleEmitter.EmitParticles();
30
31   // Update bounding shapes
32   AABB, Sphere = CalculateBoundingShapes();
33
34   // Copy this updated data to the concurrent particle data object
```

```
35    copiedAtts.particleConcurrentParticleData.CommitParticleData
36                                              (particleArray,
37                                              newParticleIndicesArray,
38                                              AABB, Sphere);
39  }
```

Code Listing 6 illustrates how the function, CommitParticleData, copies data from the background task to the *Concurrent Particle Data* instance. The function copies all of the particles in the `Particle Array`, and only a subset of information that is related to drawing particles is copied per particle. It then appends the indices of the particles that were just newly emitted to its member array, `newParticleIndices`. At the end of the function, it copies the *Bounding Objects*. During all of this, the *Concurrent Particle Data* instance is locked by locking a member variable mutex.

Code Listing 6: Pseudocode describing what CommitParticleData does when it copies its data to the *Concurrent Particle Data* instance.

```
1   ConcurrentParticleData::CommitParticleData(particleArray,
2                                              newParticleIndicesArray,
3                                              AABB, Sphere)
4   {
5     // Lock the mutex for the scope of the whole function
6     using lock(this.mutex)
7     {
8       // Copy over all particles to the member particle array
9       for(i = 0 to particleArray.size())
10      {
11        this.particleArray[i] = particleArray[i].copySubset();
12      }
13
14      // Append the new indices to the member new indices array
15      this.newParticleIndices.append( newParticleIndices.copy() );
16
17      // Copy the bounding information to the member variables
18      this.AABB = AABB.copy();
19      this.Sphere = Sphere.copy();
20    }
21  }
```

2.3.3.4 Render Phase

Unlike the **Update** phase, the **Render** phase is not directly called by the *Particle Effect Component*. Instead, it is called by the *Renderer* application system. The *Renderer* retrieves the active scene from the *State Manager*, filters entities added to the scene that have Render Components, and attempts to render them.

The Render function uses the *Drawable* instance to draw the *Particle Effect Component*. The *Drawable*, in turn, uses *Concurrent Particle Data* to read the information needed about the particle effect in order to render it. The Draw function within the *Drawable* first locks the *Concurrent Particle Data* instance using a Lock function, and then proceeds to work with the copied data in *Concurrent Particle Data*. It uses the newParticleIndices to register newly emitted particles to the *Static Billboard* that is used for rendering, and continues on to iterate over the copied array of particles in order to draw each particle that is currently active. When it is finished, it unlocks the *Concurrent Particle Data* instance.

Code Listing 7: Pseudocode showing how a *Particle Effect Component* uses its *Drawable* to render itself.

```
1  ParticleEffectComponent::Render(camera)
2  {
3    // Use the Particle Drawable to draw particles to the screen
4    if(this.playbackState is playing or this.playbackState is stopping)
5      this.drawable->Draw(camera);
6  }
7
8  Drawable::Draw(camera)
9  {
10   // Lock the Concurrent Particle Data object
11   this.concurrentParticleData.Lock();
12
13   // Activate the newly emitted particles in the static billboard
14   for(particleIndex in this.concurrentParticleData.GetNewParticleIndices())
15   {
16     ActivateParticleInBillboard(particleIndex);
17   }
18
19   // Iterate through all of the particles and draw each one
20   DrawParticles(this.concurrentParticleData.GetParticleArray(), camera);
21
```

```
22    // Unlock the Concurrent particle Data object
23    this.concurrentParticleData.Unlock();
24  }
```

2.3.3.5 Clean Up Phase

Of all phases, this is one is by far the simplest. It releases all resources and clears allocated arrays. Specifically, it cleans up the `Particle Array`, the *`Concurrent Particle Data`* instance, the *`Drawable`* instance, the *`Emitter`* instance, and the array of `Affectors`.

2.3.4 Usage

Creating a basic scene that uses a *`Particle Effect Component`* in ChilliSource is fairly simple. As Code Listing 8 illustrates, it requires that we first create a camera along with a light source and add them both to the current *`State`*'s *`Scene`*. Utilizing ChilliSource's API to create an entity is easy to use and understand, as Code Listing 9 demonstrates[3]. The *`Basic Entity Factory`* used in Code Listing 8 is not built-in to ChilliSource, and it was created to abstract the creation of entities like rooms, lights, and cameras.

Code Listing 8: Creating a scene that adds a basic *`Particle Effect Component`* in ChilliSource.

```
1   void State::OnInit()
2   {
3     // Retrieve the resource pool application system to create a particle effect
4     // Set the state's current scene's clear color to black
5     GetScene()->SetClearColour(CS::Colour::k_black);
6
7     // Retrieve a state system that will help create basic entities like
8     // cameras, rooms, and lights.
9     auto basicEntityFactory = CS::Application::Get()
10                               ->GetSystem<Common::BasicEntityFactory>();
11
12    // Create a basic room and add it to the scene
13    CS::EntitySPtr room = basicEntityFactory->CreateRoom();
```

[3] This is the only time within this paper that pseudocode will not be used. Since we are demonstrating how to use and create a *`Particle Effect Component`*, it seemed appropriate to show the code verbatim.

```
14    room->GetTransform().SetPosition(0.0f, 10.0f, 0.0f);
15    GetScene()->Add(room);
16
17    // Create a third person camera and add it to the scene
18    auto camera = basicEntityFactory
19                ->CreateThirdPersonCamera(room, CS::Vector3(0.0f, -9.0f, 0.0f));
20    GetScene()->Add(std::move(camera));
21
22    // Create an ambient light and add it to the scene
23    CS::EntitySPtr ambientLight = basicEntityFactory
24                                ->CreateAmbientLight
25                                (CS::Colour(0.65f, 0.65f, 0.65f, 1.0f));
26    GetScene()->Add(ambientLight);
27
28    // Create a particle effect entity and add it to the scene
29    auto particleEntity = CreateParticleEffect();
30    GetScene()->Add(std::move(particleEntity));
31  }
```

Code Listing 9: The function to create an ambient light with the `Basic Entity Factory` state system.

```
1   CS::EntityUPtr BasicEntityFactory::CreateAmbientLight
2                                                         (const CS::Colour& in_colour)
3   {
4     // Use the application system, Render Component Factory, to create an
5     // ambient light.
6     auto ambientLightComponent = m_renderComponentFactory
7                                 ->CreateAmbientLightComponent(in_colour);
8
9     // Create an entity, and attach the ambient light component to it.
10    auto entity = CS::Entity::Create();
11    entity->AddComponent(ambientLightComponent);
12
13    // Return the entity
14    return entity;
15  }
```

Once we've added a room, lighting, and a camera to our scene, we create a particle effect entity using a function called CreateParticleEffect. The exact code inside of this function is shown in Code Listing 10. Inside of this function, we use the main *Application* instance to generate a *Particle Effect* from a csparticle file, and it accomplishes this by using a *Resource Pool* application system. Once we have a *Particle Effect*, we can use a *Render Component Factory* application system to create a *Particle*

`Effect Component` using the newly created `Particle Effect`. Lastly, we must add the `Particle Effect Component` to an entity and add that entity to the current state's scene.

Code Listing 10: Creating a basic particle effect component with ChilliSource.

```
CS::EntityUPtr State::CreateParticleEffect()
{
  // Retrieve the singleton application instance through a static method
  auto app = CS::Application::Get();

  // Retrieve the resource pool application system to create a particle effect
  // from a JSON csparticle file
  auto resourcePool = app->GetResourcePool();
  auto particleEffect = resourcePool
                    ->LoadResource<CS::ParticleEffect>
                    (CS::StorageLocation::k_package,
                        "Particle/generic.csparticle");

  // Using a render component factory application system and the particle
  // effect, create a particle effect component
  auto renderComponentFactory = app->GetSystem<CS::RenderComponentFactory>();
  auto particleComponent = renderComponentFactory
                        ->CreateParticleEffectComponent(particleEffect);

  // Set the particle component's playback to looping
  particleComponent
      ->SetPlaybackType(CS::ParticleEffectComponent::PlaybackType::k_looping);

  // Create an entity, and add the particle component to the entity
  auto particleEntity = CS::Entity::Create();
  particleEntity->AddComponent(std::move(particleComponent));

  return particleEntity;
}
```

Compared to Reeves' particle systems, we can observe here that a `Particle Effect Component` is not composed of a hierarchy of child `Particle Effect Components`. As mentioned in Section 2.2.2, however, `Entities` are composed of `Components` and child `Entities`. Thus, ChilliSource's architecture allows the developer to create parent and child `Particle Effect Components` by nesting `Entities`.

CHAPTER 3 THE CHILLISOURCE AUTOMATED PONG GAME

3.1 Overview

The ChilliSource Automated Pong Game [4] (i.e. CSAPong) is the application that I built to aide in studying ChilliSource's `Particle Effect Components`. CSAPong is a derivative of CSPong [5], a simple pong game created with ChilliSource that was developed by the ChilliWorks team. CSPong was built to illustrate as much of the engine as possible without creating a complex game, and the three main states attempt to reflect this goal. These states include the `Splash State`, the `Main Menu State`, and the `Game State`. These states demonstrate the engine's life cycle events, creation and utilization of custom state systems and custom events, usage of the UI app system and the `State Manager`, and, most importantly, how to design entities and components to add enemy AI, player interaction, physics, collision detection, and so on. All three states are shown in Figure 3.1.

Unlike CSPong, CSAPong only uses a single state, the `Game State`. The main application instance restarts the `Game State` a number of times, and it automatically quits the application when it's finished. CSAPong does not have enemy AI or player interaction (i.e. enemy and player paddles), but the walls do collide with the ball and keep track of the "score". Additionally, CSAPong attaches a `Particle Effect Component` to the ball that will emit particles as the ball moves around the arena. Figures 3.3, 3.4, and 3.5 show a variety of examples of CSAPong in action. These examples are explained in more detail later on in this chapter.

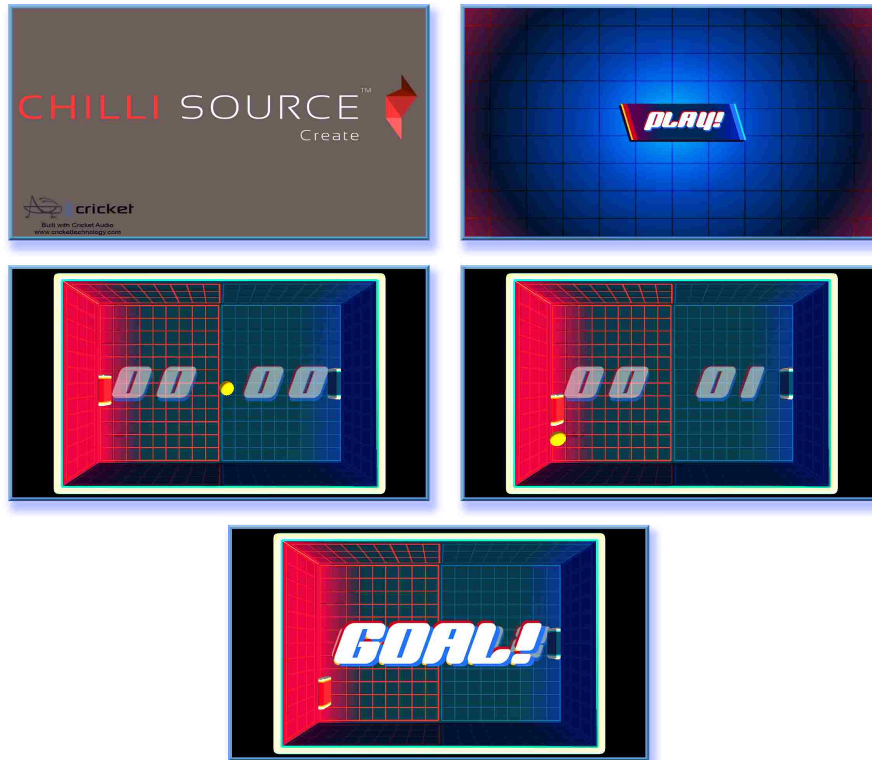

Figure 3.1: The above screenshots illustrate the original pong game. The screenshots were taken from the beginning of the application to the first goal. The top-left screenshot is from the *Splash State*, the top-right screenshot is from the `Main Menu State`, and the other three screenshots are from the *Game State*.

3.2 Architecture

3.2.1 Overview

Although it is not necessary to go over every single class that makes up CSAPong, it is essential to understand its general architecture. As shown in Figure 3.2, we can broadly describe the structure having two main parts, the ChilliSource engine[1] and the CSAPong application itself. Further, CSAPong can be defined as having *Application Systems*, a *Game State*, and the *Game State*'s *State Systems*.

[1]Obviously, we know that the entirety of the ChilliSource engine is part of the CSAPong application, but we will include it in our discussion of CSAPong's architecture due to the custom application systems that we created within ChilliSource specifically for CSAPong.

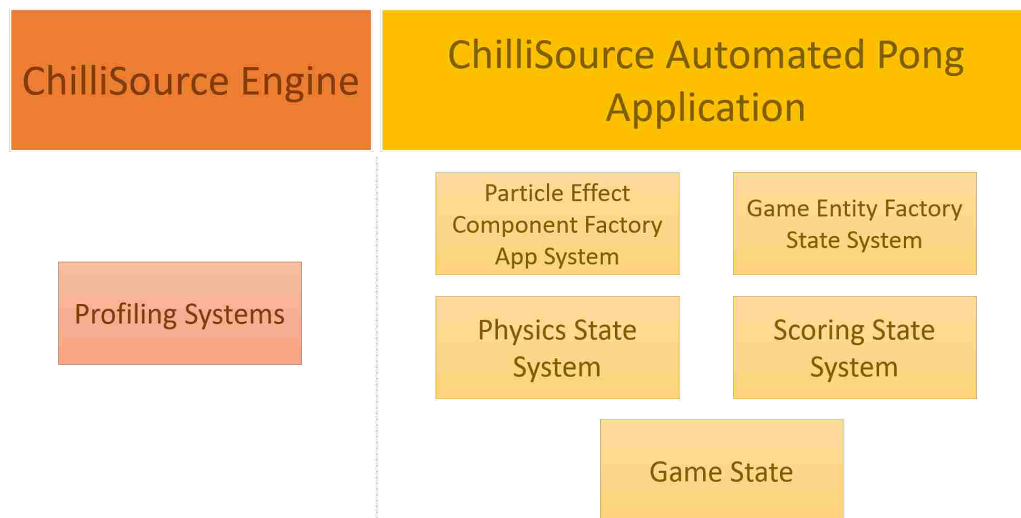

Figure 3.2: The basic architecture of the automated pong game. This shows CSAPong having a *Particle Effect Component Factory App System*, a *Game Entity State System*, a *Physics State System*, a *Scoring State System*, and the *Game State* itself. Although the entirety of the ChilliSource engine is used in CSAPong, this graph highlights the custom *Profiling Systems* within ChilliSource that I created for the purposes of instrumenting *Particle Effect Components*.

3.2.2 CSAPong Application

The *CSAPong Application* instance creates all application systems that will have a lifetime lasting throughout the whole application. This includes *Particle Effect Component Factory* and *Profiling* application systems, where the former is defined within CSAPong and the latter is defined within ChilliSource itself. It also handles pushing the initial *Game State*, restarting the *Game State*, and quitting the application.

3.2.3 Application Systems

Although not explicitly part of CSAPong, the *Profiling Application Systems* are integral to our studies. They are application systems I created that help keep track of engine-specific metrics such as particles per emission, particles actually emitted, particles actually rendered, time spent in `ParticleUpdateTask`, time spent in `CommitParticleData`, and so forth. The *Profiling Application Systems* are discussed in more detail in Appendix A.

The `Particle Effect Component Factory Application System` generates `Particle Effect Components` and attaches them to entities. The user can pass it an entity, a csparticle file path, and the number of desired `Particle Effect Component(s)` to add, which it uses to add the desired `Particle Effect Component(s)` to the entity. This application system was used to generate different kinds of particle effects as CSAPong ran.

3.2.4 Game State

The `Game State` manages the game's life cycle, its entities and components, and its user interface. Similar to the main application instance, the `Game State` creates all state systems that will have the same lifetime as the state. It uses the `Game Entity Factory State System` on initialization to create a camera, some lighting, the arena, and the ball. The `Game Entity Factory`, in turn, uses the `Particle Effect Component Factory` to attach particle effects to the ball during that ball's creation. It also signals the `Profiling Application Systems` to start gathering metrics once the `Game State` is fully initialized and has started the game.

3.2.5 State Systems

The `Physics State System` handles the movement of dynamic bodies (such as the ball) and collision checking. This behavior is described with components that are attached to entities which is in the spirit of ChilliSource's composition-based design. Thus, if we want the ball to be able to move, we attach a special type of component to the ball, a `Dynamic Body Component`, which is used by the `Physics State System`. Additionally, if we want collisions to occur between dynamic bodies and the walls, we must add static body components to the four wall entities within the arena.

The `Scoring State System` uses the collision detection in the `Physics State System` to trigger points being added to each "opposing team". As we can see in Fig-

ure 3.3, however, there is a small bug in which 2 points are awarded each time the ball collides with the wall. The bug did not interfere with studying the `Particle Effect Component`, and so time was not taken to fix it.

3.3 Gameplay

3.3.1 Logistics

The gameplay can be described as a series of "games" where the ball begins at the center of the arena, accelerates toward the bottom right corner, bounces to the right-most wall, bounces to the center of the top wall, and heads toward the center of the left-most wall. Although the game can continue for longer than this, the majority of our studies set the "game length", or ***run time***, to 5 seconds, and so this is the most frequent behavior observed. This ball path was specifically chosen to minimize the number of particles that would overlap each other during a game in order to challenge the engine to render as many particles as possible. If the particles did overlap, then those particles would be culled, or not drawn, resulting in the engine working less intensively.

Another variable that can be changed is the number of "games", or the ***maximum run number***, that will be "played". The game has the ability to change particle effect types during execution, and so the maximum run number is the number of games that are ran *for each* particle effect type that we want to use. For example, if we had 3 different particle effect types and the maximum run number was set to 3, then we would expect to see 9 games played in total.

When the "game" is over, the `CSAPong Application` instance destroys the current `Game State` and creates a new `Game State` with the same kind of particle effect type until it reaches the maximum run number. The application will then either create a new `Game State` with the next particle effect type or quit the application if there is not another particle effect type to "play". Once the application has finally quit, a series of files within a timestamped folder will be placed in the application's save data. These files contain the

instrumentation information, and we will learn more about them later on in this chapter.

3.3.2 Examples

Particle effects, as we found in the previous chapter, are complicated and have many changing variables that influence their behavior. Although many of these variables were changed during our studies, the different scenarios encountered can be generalized as the following:

- No particles emitting during the game
- A single, giant burst of particles emitting during the game
- Multiple bursts of particle emissions during the game

Figures 3.3, 3.4, and 3.5 show screenshots of these three different scenarios. The variables and configurations that drive these three scenarios are complex, (and are explained in fine detail in Appendix A), but it is important to understand the three basic variables that define these particle effects:

- Particle Min, Max, and Step
  - These are values that determine the PPE and TMP over the course of a series of games. These help the automated game create particle effects of varying PPE and TMP over time as the series of games run. These variables do not change as the series of games are ran.
- Particles Per Emission (or PPE)
  - The number of particles that will emit from the particle effect during each emission event. This changes as the series of games are ran.
- Total Maximum Particles (or TMP)
  - The pool of particles that the particle effect owns. In more technical terms, this is the size of the *`Particle Effect Component`*'s `Particle Array`. This changes as the series of games are ran.

In order to more concretely understand how these variables work, let us consider examples

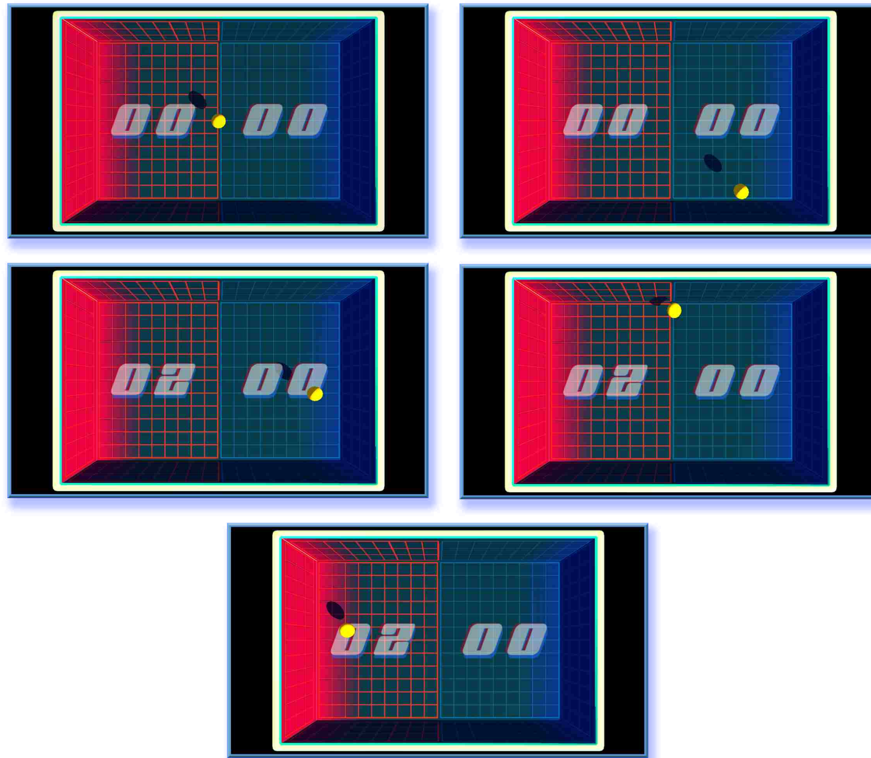

Figure 3.3: A series of screenshots showing the automated game with no particles. The screenshots were taken 1 second apart with the ball starting at the beginning of the arena.

for the single-giant-burst and multiple-burst scenarios[2].

The single-giant-burst scenario would always have PPE equal to TMP. In other words, the particle effect emits all of the particles that it has at once. An example of values for the above variables that would apply to this scenario would be:

- Particle Min, Max, and Step
  - Min: 500
  - Max: 2000
  - Step: 500
- Particles Per Emission
  - First game: 500

[2]The first scenario's variables are trivial; the PPE, TMP, and particle min/max/step would all be 0.

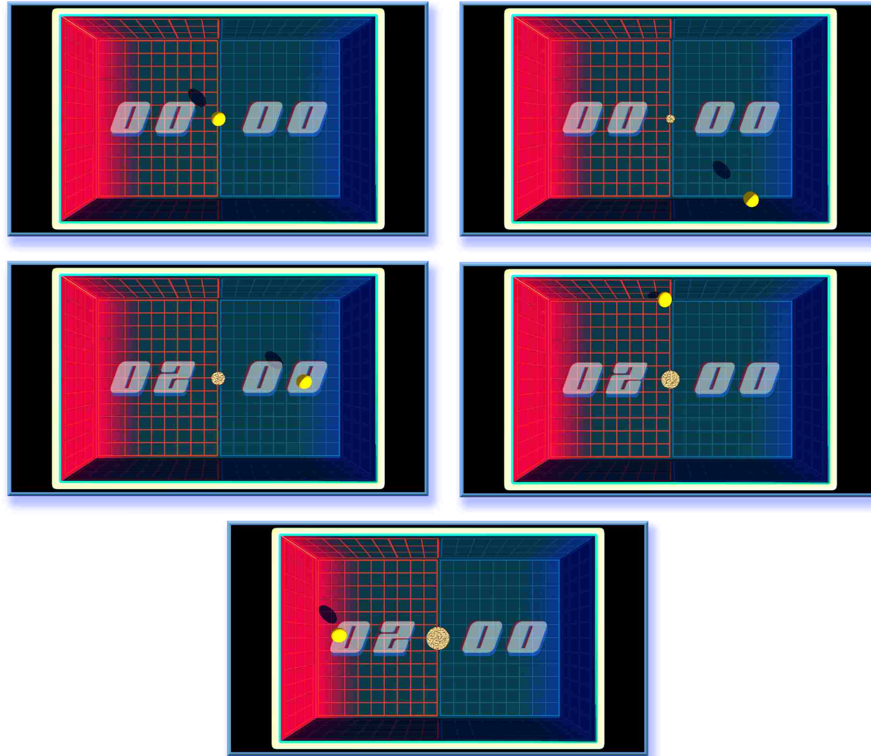

Figure 3.4: A series of screenshots showing the automated game with a single, giant burst of particles. The screenshots were taken 1 second apart with the ball starting at the beginning of the arena.

  - Second game: 1000
  - Third game: 1500
  - Fourth game: 2000
- Total Maximum Particles
  - First game: 500
  - Second game: 1000
  - Third game: 1500
  - Fourth game: 2000

The multiple-burst scenario would have the PPE be a fraction of the TMP. In other words, the particle effect emits all of its particles over time. If PPE was 10% of TMP, then the values for the main variables would be:

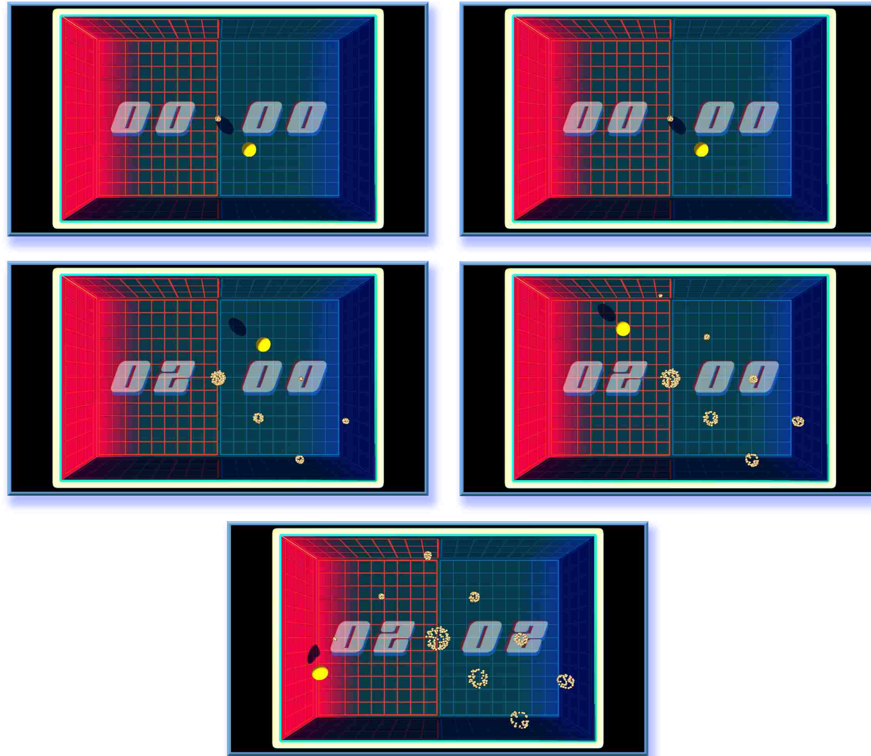

Figure 3.5: A series of screenshots showing the automated game with multiple bursts of particle emissions. The screenshots were taken 1 second apart with the ball starting at the beginning of the arena.

- Particle Min, Max, and Step
  - Min: 500
  - Max: 2000
  - Step: 500
- Particles Per Emission
  - First game: 50
  - Second game: 100
  - Third game: 150
  - Fourth game: 200
- Total Maximum Particles
  - First game: 500

  - Second game: 1000
  - Third game: 1500
  - Fourth game: 2000

In addition to keeping track of these variables, information (or metrics) about the `Particle Effect Components` are gathered during the game. Particle metrics describe information such as the number of times every particle was drawn and the number of emissions that were made. Timing metrics, on the other hand, describe how long different sections of code take. The following is a list of the two important particle metrics that are most often used in my results:

- Render ball function calls
  - This describes the number of times that the ball itself was rendered. It serves as a way to gauge how visually smooth (or how many frames are processed) the game is running.
- Particles actually rendered
  - This shows how many times every single particle across all particle effects are rendered. This also serves as a way to gauge how visually smooth the game is running.

A higher value for both *render ball function calls* and *particles actually rendered* indicate better performance. This is because, when more particles are rendered or the ball is rendered more often, the game appears to run more smoothly. Or, in other words, the "frames per second" improves.

It is unnecessary to know more than the above metrics and variables to understand the results of this thesis, but Appendix A provides much more detail for those who are motivated to learn more.

3.4 Initial Results

A variety of these automated games were run and their outputs were examined on Windows, iOS, and Android platforms. This was done in an effort to find an area within the *Particle Effect Component* class that needed optimization. In the beginning of these studies, the only available datasets were from the *Metrics System* and the Visual Studio CPU sampling profiles, as the timing instrumentation was still not complete (see Appendix A for more information). I observed that a particular automated game in which the *total maximum particles* and *particles per emission* values were the same[3] caused the game to visually lag, or skip frames. Armed with only the metrics information and CPU sampling (see Figure 3.6), all that was known was that the following *Particle Effect Component* methods had a considerable number of samples:

- `ParticleDrawable::DrawParticles`
- `ParticleUpdateTask`
- `ParticleAffector::AffectParticles`

Given that these three methods were run repeatedly throughout the games, these results were unsurprising. At the time, it was tempting to pursue `ParticleUpdateTask` since it iterated over the entire particle array multiple times. I thought that, perhaps, creating a cache-friendly data structure that kept track of active particles would be a good initial optimization. However, as it shows in Figure 3.8, the functions that iterate over all of the particles were only taking $(1.0 + 1.7 + 12.0 + 0.6)/48.4 \approx 32\%$ of `ParticleUpdateTask`'s time. The `CommitParticleData` function, on the other hand, was taking $31.5/48.4 \approx 65\%$ of `ParticleUpdateTask`'s time, and 29.4 of those 31.5 seconds were spent waiting for a lock to release. Thus, it is clear that, in this case, `ParticleUpdateTask`'s bottleneck was `CommitParticleData`.

These initial results not only demonstrated a possible place to optimize, but it also reinforced the importance of backing up possible optimizations with data in order to avoid opti-

[3] Specifically, *total maximum particles* and *particles per emission* were 10,000 in Figures 3.6, 3.7, and 3.8.

Level	Function Name	Inclusive Samples %	Exclusive Samples %
8	ChilliSource::Renderer::RenderSceneToTarget	66.57	0
9	ChilliSource::Renderer::Render	63.03	0
10	ChilliSource::ParticleEffectComponent::Render	60.84	0
11	ChilliSource::ParticleDrawable::Draw	60.7	0
12	ChilliSource::StaticBillboardParticleDrawable::DrawParticles	60.36	0
13	ChilliSource::StaticBillboardParticleDrawable::DrawWorldSpace	60.21	0.2
12	ChilliSource::TaskPool::ProcessTasks	29.02	0
13	ChilliSource::TaskPool::PerformTask	29.02	0
20	ChilliSource::`anonymous namespace'::ParticleUpdateTask	28.94	0.13
14	ChilliSource::`anonymous namespace'::BuildSpriteData	25.13	0.55
14	ChilliSource::DynamicSpriteBatch::Render	24.3	0.23
21	ChilliSource::ColourOverLifetimeParticleAffector::AffectParticles	14.02	0.32
15	ChilliSource::GenericVector3<float>::Rotate	11.39	0.52
16	ChilliSource::GenericVector3<float>::Rotate	10.81	1.12
15	ChilliSource::DynamicSpriteBatch::ForceRender	9.33	0
16	ChilliSource::DynamicSpriteBatch::BuildAndFlushBatch	9.26	0

Figure 3.6 A Visual Studio call tree sourced from the games based on the metadata in Figure 3.7. The function call tree level is ≥ 8, the function names begin with "ChilliSource::", the inclusive samples % are $\geq 6\%$, and it is sorted by the inclusive samples % in descending order. These parameters were used to ensure that the results were low-level ChilliSource function calls with a high inclusive sample percent. The functions with the most inclusive samples are either related to rendering or updating particles.

attribute	value
are particles looping	TRUE
num particle effects	1
min particles	10000
max particles	10000
particles step	500
is total max particles changing	TRUE
is particles per emission changing	TRUE
particles per emission	0
total maximum particles	0
particles per emission step	500
total maximum particles step	500
total num runs	10
duration per run	5

Figure 3.7 Metadata of a series of games that demonstrated contention in `CommitParticleData`. The important part to understand here is that these games are similar to the scenario in which there are a single, giant burst of particles. In other words, the particle effect emits all of the particles that it has at once. We also know that it only performs this for 10,000 particles since the `min particles` and `max particles` values are the same.

timed section name	time in seconds	num section calls
ParticleDrawable Draw	41.114	692
ParticleDrawable Draw Lock	0.0222	692
ParticleUpdateTask	48.4279	652
ParticleUpdateTask BoundingShapes	0.616089	652
ParticleUpdateTask CommitParticleData	31.5472	652
ParticleUpdateTask CommitParticleData Lock	29.3924	662
ParticleUpdateTask ParticleAffectors	12.0305	652
ParticleUpdateTask ParticleEmission	1.71181	652
ParticleUpdateTask ParticleIteration	0.998673	652

Figure 3.8 Timing output based on the metadata in Figure 3.7. The important timing sections to focus on are *ParticleUpdateTask CommitParticleData* and *ParticleUpdateTask CommitParticleData Lock*. The former shows 31.55 seconds, and the latter shows 29.39 seconds. This tells us that the `CommitParticleData` function spent about 29 of its 32 seconds waiting for a lock to release.

mizing into a vacuum. Without the timing data, I could have easily gone forward with optimizing particle iteration instead of focusing on the contention in `CommitParticleData`, and that would have been a waste of time.

CHAPTER 4 CHILLISOURCE PARTICLE EFFECT COMPONENT OPTIMIZATION CASE STUDIES

4.1 Counting Contention

From the initial results in the previous chapter, we found that CommitParticleData experienced a great deal of contention. Figure 3.8 showed that, of the 31.5 seconds that CommitParticleData took, 29.4 seconds was spent waiting for a lock to release. It can be inferred, then, that it only needed about 2 seconds to do any meaningful work and, if this contention could be reduced, then that function could be drastically sped up.

The results of the following studies demonstrated the differences between managing con-

Function Name	Thread Counter > 1 After Increment *(thread is waiting for lock)*	Thread Counter >1 Before Decrement *(other thread is waiting for lock)*
StartUpdate	301	0
HasActiveParticles	0	0
GetAABB	0	0
GetBoundingSphere	0	0
Lock	1	0
Unlock	0	521
CommitParticleData	521	302

Figure 4.1: The results of "counting the contention" between the background and main threads in *Concurrent Particle Data*. This data shows that the updating thread in CommitParticleData is waiting for the drawing thread to release the locked resource. The majority of these functions are either accessors/mutators or the CommitParticleData function itself, but the roles of the Lock and Unlock functions are a little enigmatic. *Concurrent Particle Data* only has a single mutex that is used between the drawing and updating threads and, since the drawing thread does not have direct access to *Concurrent Particle Data*'s mutex, the Lock and Unlock functions are there to lock and unlock the mutex in question.

tention for a single, large data structure and for many pieces or "shards" of a fragmented data structure. I found that ChilliSource's *Particle Effect Component* was likely to have contention between the background and main threads (the threads that update particles and draw particles, respectively) for a single particle effect. For many particle effects, however, there was less contention between the background and main threads since the particles were distributed (or "sharded") across many *Particle Effect Components* and the main thread had a smaller chance of contending for a given effect's particle array.

The first step in reducing this contention for these case studies was knowing for certain what was making the background thread in CommitParticleData wait. The simplest way to ascertain what was causing the contention was to do some counting. In *Concurrent Particle Data*, an atomic thread counter was used to keep track of how many threads were contending for resources. This thread counter was incremented right before a thread attempted to gain access to a lock, and it was decremented right before it relinquished access to that lock. The IncrementCounter and DecrementCounter methods printed a message when the thread counter was greater than one. During incrementation, a thread counter that was greater than one meant that the current thread was waiting to retrieve the lock. During decrementation, a thread counter that was greater than one meant that another thread was waiting for the current thread to release the lock. Since only one background thread was scheduled, we know that the thread counter should never be greater than two.

This "contention counting" was done using the same metadata from Figure 3.7 in my initial results, and the data collected can be found in Figure 4.1. The functions in which either the background or main threads waited for a lock to release were StartUpdate[1] (301 times), Lock (1 time), and CommitParticleData (521 times). The functions in which the background or main thread had the other thread waiting for a lock to release were Unlock (521 times) and CommitParticleData (302 times). These results illustrated that StartUpdate and Lock were waiting on CommitParticleData since 301+1 = 302,

and that CommitParticleData was waiting on Unlock since their counts were both 521.

With these results in mind, recall from Section 2.3.3 that the Lock function was used by the *Particle Drawable* when drawing particles in order to lock ConcurrentParticleData and secure access to the particle array. The *Particle Drawable* is the only class that uses this Lock function. The *Particle Drawable* reads information from *Concurrent Particle Data*'s particle and new particle indices array. Thus, I inferred that it was unnecessary to lock both of those data structures at once and, instead, more than one mutex in the CommitParticleData and Draw functions could be used in an attempt to reduce the contention that we observed in our initial results shown in Figure 3.8.

4.2 Using More Than One Mutex

4.2.1 Code

The changes needed to add mutexes for the two data structures and remaining variables that were copied were fairly minimal. `particlesMutex`, `newParticlesIndicesMutex`, and the `normalMutex` were added as member variables to *Concurrent Particle Data*. Each of these mutexes were locked when their respective variables needed to be read or written to. When a mutex was locked within *Concurrent Particle Data*, it was done by directly accessing the mutex (see Code Listing 11). When a mutex was locked within the *Particle Drawable*, it was done by utilizing special locking functions[2] provided by *Concurrent Particle Data* (see Code Listing 12).

One of the potential issues with these changes was that, even though a particle may currently be active, it may not be available for rendering by the time the DrawParticles function was called. To more concretely explain, imagine that the *Particle Drawable*

[1] Although not a part of the pseudocode provided for the ParticleEffectComponent::OnUpdate method in earlier chapters, this simple function merely lets the caller know whether or not particle data has been committed since the last time it was called.

[2] In actuality, a single locking function was used. An enumerator was passed to the function specifying which mutex needed to be locked. Although this was better practice, it obfuscated the actual meaning of the code by making it more verbose. Thus, a tidier version without the enumerators are used here.

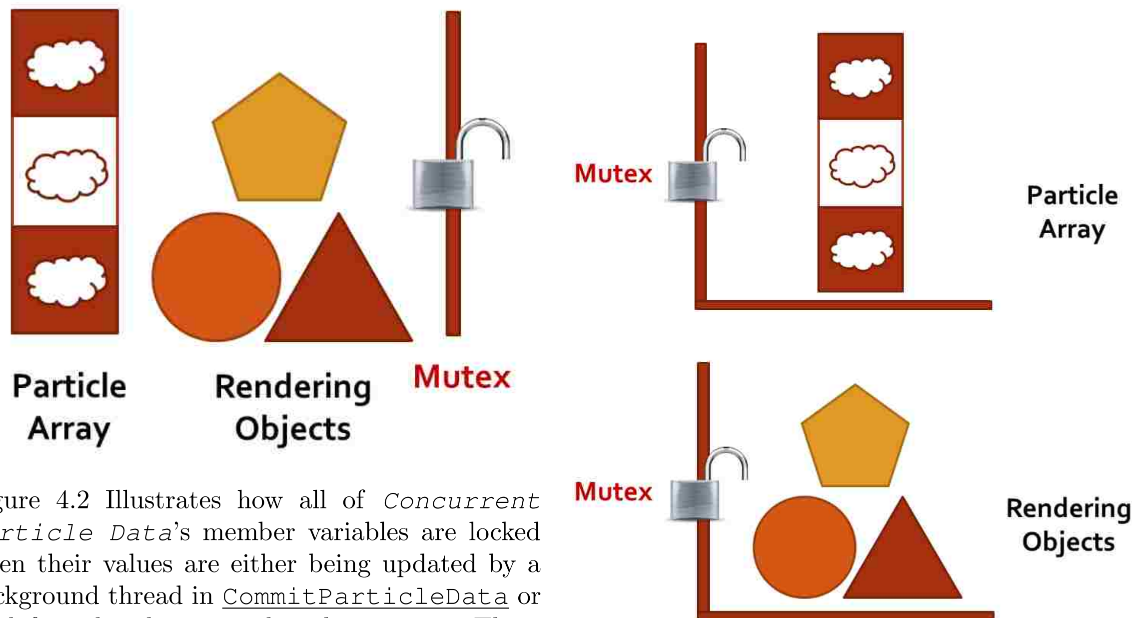

Figure 4.2 Illustrates how all of *Concurrent Particle Data*'s member variables are locked when their values are either being updated by a background thread in CommitParticleData or read from by the main thread in Draw. These member variables include the particle array, the new particle indices array, and the bounding information. In this figure, the latter variables are abstracted as "rendering objects" for simplicity and to emphasize that the most important variable to consider is the particles array. This is because the background (i.e. updating) and main (i.e. rendering) threads contend for the particles array the most out of all of the member variables.

Figure 4.3 Illustrates how the particles array and the rendering objects are given separate mutexes for the more than one mutex case study. As stated in Figure 4.2, the particles array is emphasized in this figure because it is contended for the most out of all of the member variables in *ConcurrentParticleData*.

was taking in new particle indices to activate within the static billboard to make it available for rendering. At the same time, CommitParticleData was copying new particle data and there happened to be newly emitted particle indices. Since the *Particle Drawable* had old information that did not include these newly emitted particles, these new particles would not be rendered because, even though they were marked as active, they were not activated in the static billboard. An example of this behavior is shown in Figure 4.4. Although this sounds like a significant problem, it can be argued that the inconsistent state cannot possibly be perceived by the user since a considerable number of updates happen within a short span of time. If we look at Figure 4.8, for example, the number of calls to the

Time Step	Updating Particles Thread Actions	Drawing Thread Actions	New Indices Contents	Particles Activated in Billboard
0	---	---	[1, 2, 3]	[]
1	Particles Lock: Particles are updated.	New Indicies Lock: Particles 1 – 3 are cleared from array and activated in billboard.	[]	[1, 2, 3]
2	New Indices Lock: [4, 5, 6] are added.	Particles Lock: Particles are rendered, but [4, 5, 6] are ignored.	[4, 5, 6]	[1, 2, 3]
3	Particles Lock: Particles are updated.	New Indicies Lock: Particles [4, 5, 6] are cleared from array and activated in billboard.	[]	[1, 2, 3, 4, 5, 6]
4	New Indices Lock: Nothing is added.	Particles Lock: Particles are rendered.	[]	[1, 2, 3, 4, 5, 6]
5	---	New Indices Lock: Nothing is activated.	[]	[1, 2, 3, 4, 5, 6]
6	Particles Lock: Particles are updated.	Wants particle lock, but has to wait.	[]	[1, 2, 3, 4, 5, 6]
7	New Indices Lock: [7, 8, 9] are added.	Particle Lock: Particle are rendered, but [7, 8, 9] are ignored.	[7, 8, 9]	[1, 2, 3, 4, 5, 6]

Figure 4.4: Illustrates the relationship between locking two mutexes between the drawing and updating threads over the lifetime of a particle effect. This shows how active particles in the particle array may not be rendered immediately because they were not activated in the Drawable's billboard before rendering. This can happen if the "updating particles" thread is holding the particles lock and copying particles in `CommitParticleData` while the Drawable is clearing and activating "old" new particle indices from `CommitParticleData`. The array does not contain the "new" particle indices from the most recent update because those are not copied until after the particles mutex is released, as shown in Code Listing 11. It is important to emphasize that, every time a set of particle indices were ignored during rendering, it was because it was not in the Drawable's billboard, and this is shown above.

`Draw` function was about 800, and that was within approximately a 20 $runs * 5\ {}^{seconds}/_{run} =$ 100 second timespan. This meant that this function was called about 8 times per second, and it is highly doubtful that a person could notice the inconsistent state of the particle effect within 1/8th to 1/4th of a second. Thus, the temporary inconsistent state was not considered a problem.

Code Listing 11: Pseudocode illustrating the changes needed to utilize more than one mutex in the `CommitParticleData` function. Note that, instead of locking only one mutex, three different mutexes were locked at different times depending on the resource needed (i.e. the particles array, the new particle indices array, or the non-data structure array).

```
1  ConcurrentParticleData::CommitParticleData(particleArray,
2                                             newParticleIndicesArray,
```

```
                                                AABB, Sphere)
{
  // NEW -> Lock the particles mutex
  using lock(this.particlesMutex)
  {
    // Copy over all particles to the member particle array
    for(i = 0 to particleArray.size())
    {
      this.particleArray[i] = particleArray[i].copySubset();
    }
  }

  // NEW -> Lock the new particle indices mutex
  using lock(this.newParticleIndicesMutex)
  {
    // Append the new indices to the member new indices array
    this.newParticleIndices.append( newParticleIndices.copy() );
  }

  // NEW -> Lock the normal (non-data structure) mutex
  using lock(this.normalMutex)
  {
    // Copy the bounding information to the member variables
    this.AABB = AABB.copy();
    this.Sphere = Sphere.copy();
  }
}
```

Code Listing 12: Pseudocode illustrating the changes needed to utilize more than one mutex in the Draw function. Note that, instead of locking only one mutex, two different mutexes were locked at different times depending on the resource needed (i.e. the particles array or the new particle indices array).

```
Drawable::Draw(camera)
{
  // NEW -> Lock the particle new indices array
  this.concurrentParticleData.LockNewParticleIndices();

  // Activate the newly emitted particles in the static billboard
  for(particleIndex in this.concurrentParticleData.GetNewParticleIndices())
  {
    ActivateParticleInBillboard(particleIndex);
  }

  // NEW -> Unlock the particle new indices array
  this.concurrentParticleData.UnlockParticleIndices();

  // NEW -> Lock the particles array
  this.concurrentParticleData.LockParticles();

  // Iterate through all of the particles and draw each one
  DrawParticles(this.concurrentParticleData.GetParticleArray(), camera);

  // NEW -> Unlock the particles array
  this.concurrentParticleData.UnlockParticles();
```

```
23 }
```

4.2.2 Results

attribute	1 PE, 0-50K	1 PE, 50K
are particles looping	TRUE	TRUE
num particle effects	1	1
min particles	0	50K
max particles	50K	50K
particles step	5K	5K
is total max particles changing	TRUE	TRUE
is particles per emission changing	TRUE	TRUE
particles per emission	0	0
total maximum particles	0	0
particles per emission step	{500, 5K}	{500, 5K}
total maximum particles step	5K	5K
total num runs	5	20
duration per run	5	5

Figure 4.5: The metadata used for the 8 series of automated games for the case studies.

The following four figures are the result of running a series of automated games in two scenarios:

1. Scenario used for particle metrics (Figure 4.6 and Figure 4.7)
    - Min: 0
    - Max: 50,000
    - Step: 5,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
2. Scenario used for timing metrics (Figure 4.8 and Figure 4.9)
    - Min: 50,000

- Max: 50,000
- Each game was 5 seconds long
- This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.

When examining these results[3], the yellow "Original 0.1 TMP" series should be directly compared with the blue "> 1 Mutex 0.1 TMP" series and the orange "Original 1.0 TMP" series should be directly compared with the green "> 1 Mutex 1.0 TMP" series. This is because comparing the two series that "played the same game" will yield more accurate conclusions than cross-comparisons would.

4.2.3 Discussion

The figures that focus on the particle metrics *render ball function calls* and *particles actually rendered* (Figures 4.6 and 4.7) do not exhibit a great deal of improvement from the original. For both of those figures, improvement would mean higher values. This is because, when more particles are rendered or the ball is rendered more often, the game appears to run more smoothly. Or, in other words, the "frames per second" improves.

The figures focusing on the particle timing, on the other hand, show considerable improvement (Figures 4.8 and 4.9). For 0.1 TMP, `CommitParticleData` went down from 47 seconds to 24. For 1.0 TMP, `CommitParticleData` went down from 50 seconds to 31. This was approximately a 40% increase in speed![4]However, there was still a large amount of contention for the particle array. For 0.1 TMP, 19 of those 24 seconds was spent waiting for the particle array's mutex to unlock. For 1.0 TMP, 29 of those 31 seconds was spent waiting for the particle array's mutex to unlock. It was clear, then, that more must be done to reduce contention for the particle array. Instead of adding more mutexes, one could be taken away.

[3]All of the results presented in this thesis can be found in an HDF5 file hosted on a Bitbucket repository [6].
[4]See Figure 4.24 in Section 4.4 for a summary table of these timed sections.

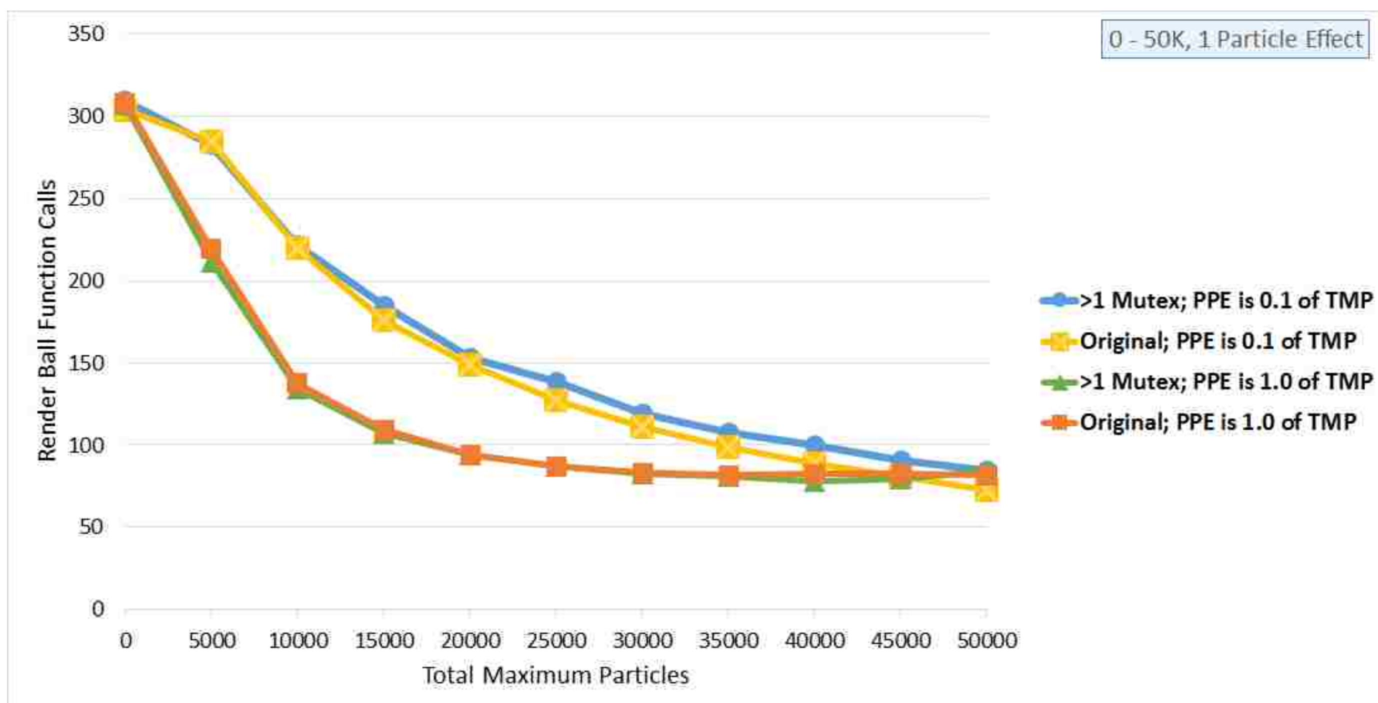

Figure 4.6 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows that changing `Concurrent Particle Data` to use multiple mutexes does not impact the number of times that the ball is rendered. The multiple-burst and single-burst (0.1 and 1.0 TMP, respectively) scenarios for the > 1 Mutex data series are almost the same as the ones for the data series without any changes to the source code (i.e. the Original data series).

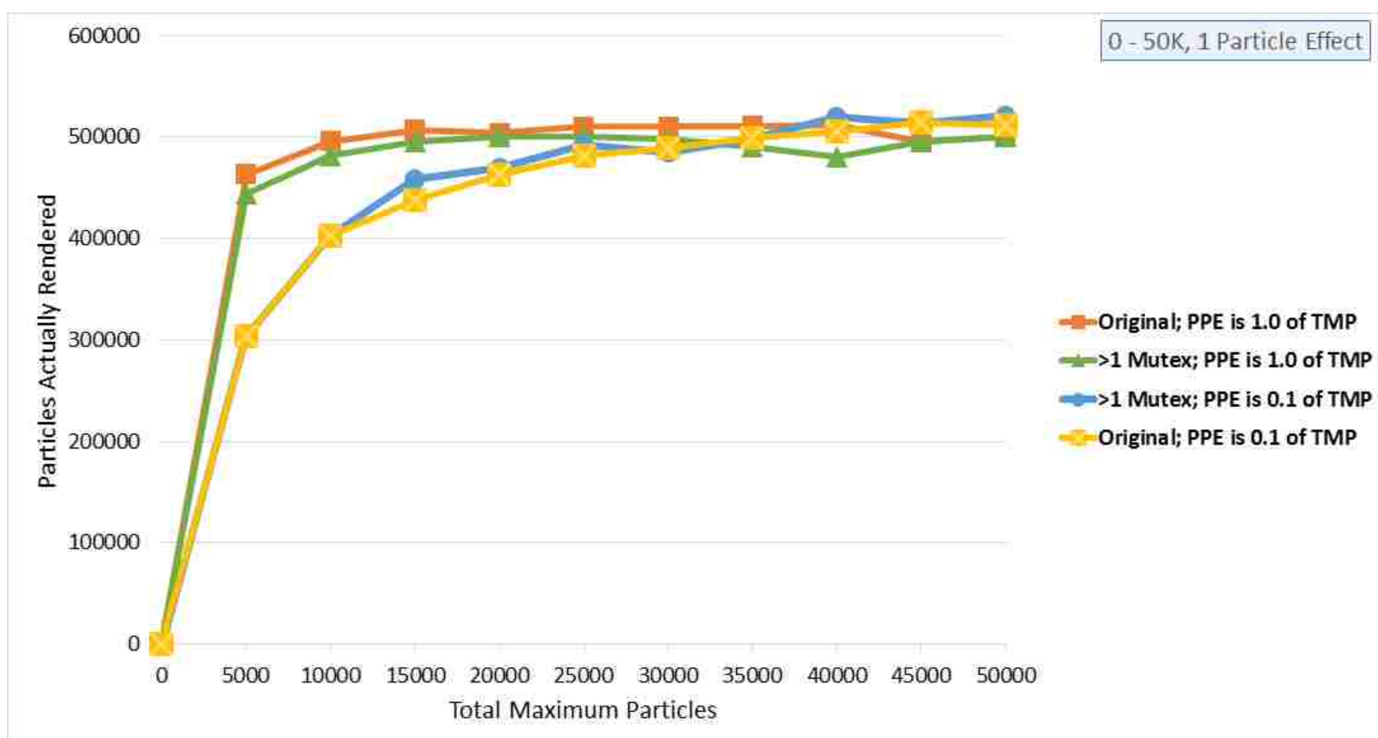

Figure 4.7 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. This shows that changing `Concurrent Particle Data` to use multiple mutexes also does not impact the number of times that particles are rendered. As in Figure 4.6, the > 1 Mutex data series is almost the same as the ones for the data series without any changes to the source code (i.e. the Original data series).

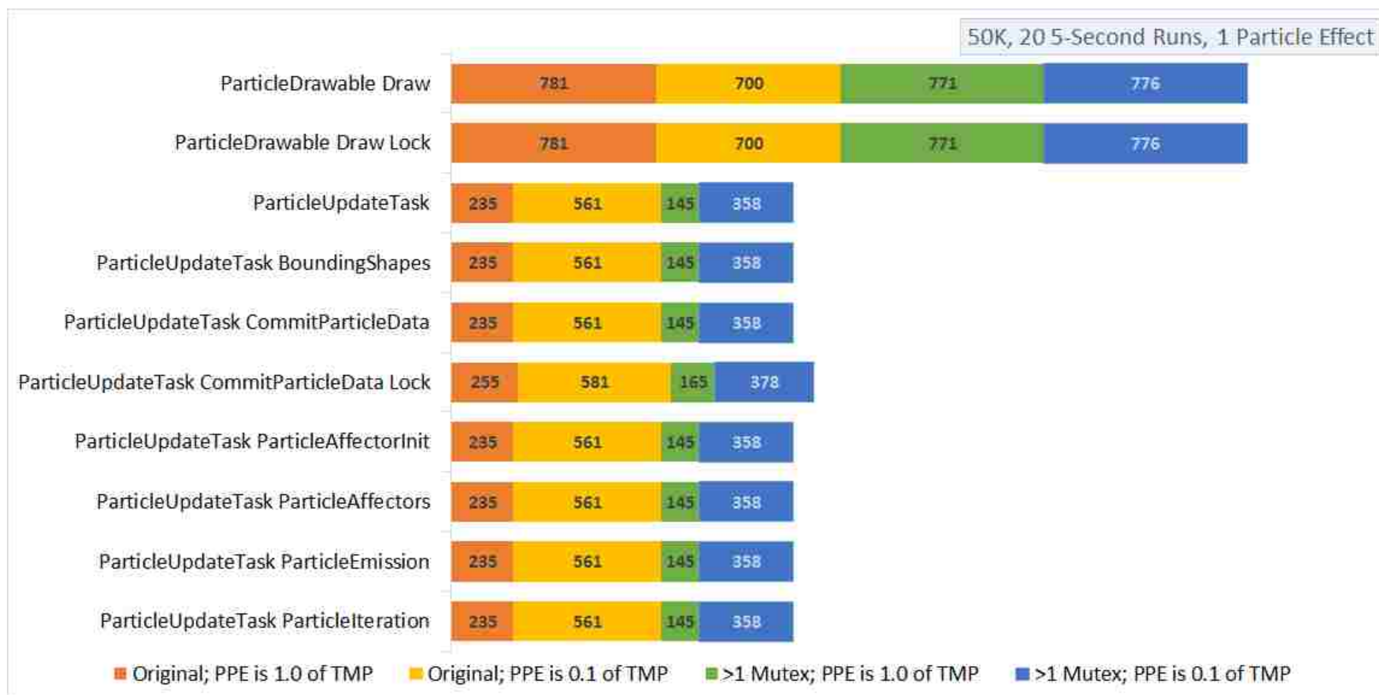

Figure 4.8 A graph showing the number of section calls for various sections of code for a single particle effect. Note that the "ParticleUpdateTask CommitParticleData Lock" section is timing the particle array lock for the > 1 Mutex data series. This shows that, between the > 1 Mutex and Original data series, the drawing functions had approximately the same number of section calls. The Original data series for the particle update functions, however, show more calls.

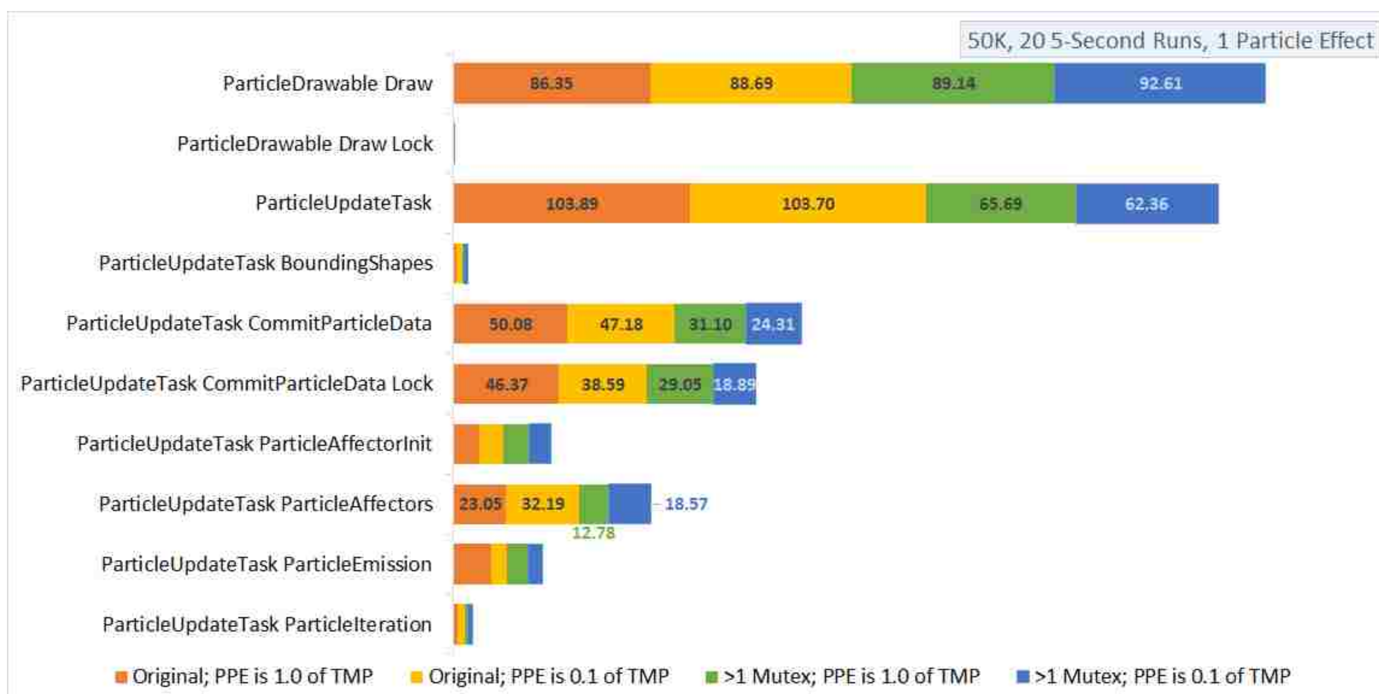

Figure 4.9 A graph showing the time spent in various sections of code for a single particle effect. Note that the "ParticleUpdateTask CommitParticleData Lock" section is timing the particle array lock for the > 1 Mutex data series. The > 1 Mutex data series shows improvement in the `CommitParticleData` function from the Original. For 0.1 TMP, `CommitParticleData` went down from 47 seconds to 24. For 1.0 TMP, `CommitParticleData` went down from 50 seconds to 31. This was approximately a 40% increase in speed.

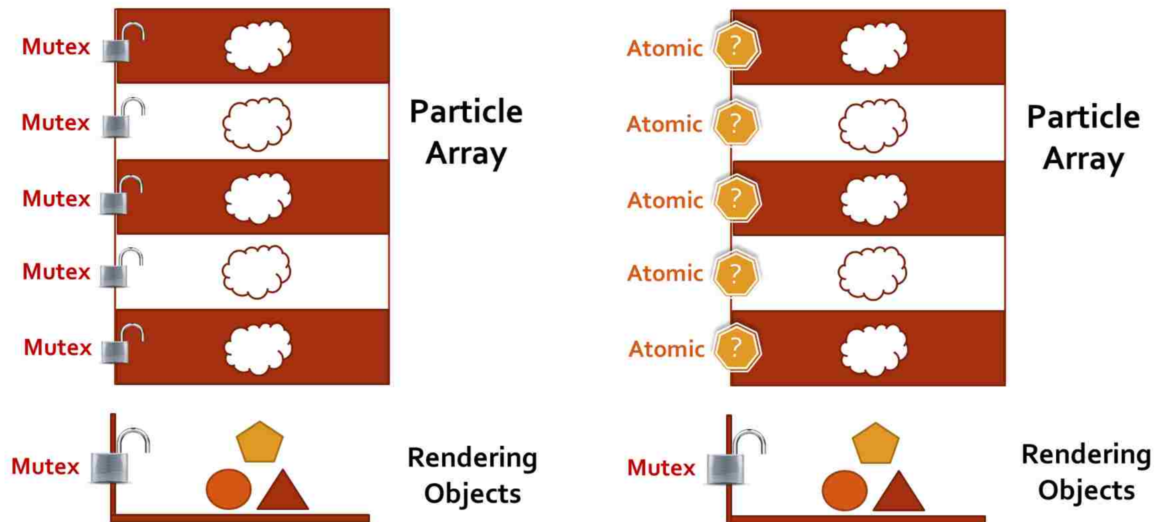

Figure 4.10: Shows the particle array with a mutex for every particle element (left) and with an atomic variable for every particle element (right). The former scenario could greatly hinder performance since mutexes have a considerable amount of overhead [7]. The latter scenario, however, transforms the particle array into a lock free data structure. This data structure has the benefit of updating and rendering threads simultaneously accessing the particle array without the overhead of mutexes.

4.3 Using A Lock Free Data Structure

Using a single mutex to protect the particle array in the previous section did reduce contention. However, if two threads wanted to access two separate particles within that array, then they would be forced to take turns since the mutex locks the entire array. In order to allow updating and rendering threads to simultaneously access the particle array, it is tempting to add a mutex for each element as shown on the left in Figure 4.10. This is not a good solution because, in general, mutexes have a considerable amount of overhead [7]. Given that the particle array may have tens of thousands of elements, attaching an expensive mutex to each element would most likely worsen performance instead of improve it. A lock free data structure, on the other hand, could allow multiple threads to access the particles array safely without the cost of mutexes.

Similar to the "multiple mutex" solution, using a lock free data structure has the potential problem of the particle array being in an inconsistent state during rendering. However, an

argument analogous to the one made above in the multiple mutex solution can be used. Even though the particle array might have partially up to date information during any given render, the renderer was going to update again very shortly and the particle array should be up to date by then. We know this because `CommitParticleData` was called frequently (see Figure 4.8) and, without waiting for locks to release, `CommitParticleData` would run fairly quickly. Therefore, the potentially inconsistent state of particles should not be noticeable to the end user.

4.3.1 Code

To make the particle array a lock free data structure, which threads are reading and which ones are writing must be identified. A "reader" thread only wants to know the value of a contended resource, and a "writer" thread wants to modify the value of a contended resource. This is an important distinction to make because many reader threads can access a contended resource simultaneously, but anytime a writer thread wants to modify the resource's value it must have exclusive access.

We know that the main thread (the one that is drawing) is only reading the particles, and we also know that the background thread is writing to the particles. Without a mutex guarding the particle array, there is a possibility that the main thread reads a particle just as the background thread is writing to it; this may result in corrupted data. Thus, a solution in which the two threads could both simultaneously access the array but not the same particle was needed.

There were a variety of strategies that could have been employed, but the simplest was to rely on an atomic instruction to achieve synchronization. Specifically, the ***compare-and-swap*** operation was used. ***Compare-and-swap*** is an atomic read-modify-write operation on a memory location that can only be implemented using special hardware instructions [8]. In one atomic action, it compares the value of a variable with what is expected and, if the variable has that expected value, it swaps that expected value with the desired value

(see Code Listing 13). Once the atomic operation is attempted, it returns a boolean indicating whether or not the operation is successful. The operation will not be successful if there is another writer using the contended resource. A ***compare-and-swap*** operation is typically wrapped within a while loop since it may return false if a concurrent or spurious write occurs. The latter may occur if the value of the variable is actually the expected value (i.e. the resource is not contended) but the operation fails anyway. This could happen if we utilized the weak version of *atomic_compare_exchange* [9], which is a ***compare-and-swap*** operation defined in the `std::atomic` library in C++11. The strong version of *atomic_compare_exchange* could be used in order to avoid spurious writes, but the checks needed within the strong version are expensive. The weak version can yield better performance if the operation is wrapped within a loop [9]. Given that this operation was supported on all of the platforms that ChilliSource supported, the weak version of *atomic_compare_exchange* was used. Due to the infrequency of spurious writes and the small number of threads, the weak version would be the best option in this case.

Code Listing 13: Pseudocode illustrating the behavior that a compare-and-swap operation. Although it is shown as a function here to demonstrate how it works, it is executed atomically via special hardware instructions.

```
1  bool compareAndSwap(contendedResource, expectedValue, newValue)
2  {
3      // Executes atomically
4      if(contendedResource is not expectedValue)
5          return false;
6  
7      // Its value was the expected value, so change it to newValue
8      contendedResource = newValue;
9      return true;
10 }
```

In order to take advantage of the *atomic_compare_exchange* function, an atomic variable must be used. An atomic boolean was incorporated into `Concurrent Particle Data`'s particle struct, as shown in Code Listing 14. This atomic boolean, `isContended`, serves as the variable that the ***compare-and-swap*** operation examines. It indicates whether or not another thread is using this particle struct. When a thread wants access to the particle

struct, it calls the *atomic_compare_exchange* function. It then compares `isContended` with "false" and, if it is false, will swap `isContended` with "true". The normal particle operations will take place, and then `isContended` will be set back to false. Code listings 15 and 16 illustrate the needed changes in the CommitParticleData and Draw functions.

Code Listing 14: The struct that is used to within the particle array in *Concurrent Particle Data* during the Lock Free case study.

```
struct Particle final
{
    // NEW -> Atomic boolean
    mutable std::atomic<bool> m_isContended = { false };
    bool m_isActive = false;
    Vector3 m_position;
    Vector2 m_scale = Vector2::k_zero;
    f32 m_rotation = 0.0f;
    Colour m_colour = Colour::k_white;
};
```

Code Listing 15: Pseudocode illustrating the changes needed to make the `particles array` a lock free data structure in the CommitParticleData function.

```
ConcurrentParticleData::CommitParticleData(particleArray,
                                           newParticleIndicesArray,
                                           AABB, Sphere)
{
  // Copy over all particles to the member particle array
  for(i = 0 to particleArray.size())
  {
    // NEW -> Retrieve the particle, but we don't have ownership of it yet
    currentParticle = particleArray[i];
    gotOwnershipOfParticle = false;

    // NEW -> Keep spinning if *isContended* is true.
    // Set it to true once it is false.
    while(gotOwnershipOfParticle is false)
    {
      gotOwnershipOfParticle = currParticle.isContended
                                    .CompareExchange(false, true);
    }

    // Copy the particle now that we have ownership of it
    this.particleArray[i] = currentParticle.copySubset();

    // NEW -> Release the particle
    currentParticle.isContended = false;
  }

  // NEW -> Lock the mutex for the remainder of the function
  using lock(this.mutex)
  {
```

```
30      // Append the new indices to the member new indices array
31      this.newParticleIndices.append( newParticleIndices.copy() );
32
33      // Copy the bounding information to the member variables
34      this.AABB = AABB.copy();
35      this.Sphere = Sphere.copy();
36    }
37  }
```

Code Listing 16: Pseudocode illustrating the changes needed to make the `particles array` a lock free data structure in the `Draw` function.

```
1   Drawable::Draw(camera)
2   {
3     // NEW -> Lock the concurrent particle data object only for new indices
4     this.concurrentParticleData.Lock();
5
6     // Activate the newly emitted particles in the static billboard
7     for(particleIndex in this.concurrentParticleData.GetNewParticleIndices())
8     {
9       ActivateParticleInBillboard(particleIndex);
10    }
11
12    // NEW -> Unlock the concurrent particle data object only for new indices
13    this.concurrentParticleData.Unlock();
14
15    // Iterate through all of the particles and draw each one
16    DrawParticles(this.concurrentParticleData.GetParticleArray(), camera);
17  }
18
19  StaticBillboardDrawable::DrawParticles(particleArray, camera)
20  {
21    // Copy over all particles to the member particle array
22    for(i = 0 to particleArray.size())
23    {
24      // NEW -> Retrieve the particle, but we don't have ownership of it yet
25      currentParticle = particleArray[i];
26      gotOwnershipOfParticle = false;
27
28      // NEW -> Keep spinning if *isContended* is true.
29      // Set it to true once it is false.
30      while(gotOwnershipOfParticle is false)
31      {
32        gotOwnershipOfParticle = currParticle.isContended
33                                            .CompareExchange(false, true);
34      }
35
36      // Render the particle if it is active and is activated in the billboard
37      if(currentParticle.isActive and currentParticle is in particleBillboard)
38      {
39        Render(currentParticle);
40      }
41
42      // NEW -> Release the particle
```

```
43        currentParticle.isContended = false;
44      }
45    }
```

4.3.2 Results

Just as in the first case study, the lock free case study produced the following four figures running a series of automated games in two scenarios:

1. Scenario used for particle metrics (Figure 4.11 and Figure 4.12)
    - Min: 0
    - Max: 50,000
    - Step: 5,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
2. Scenario used for timing metrics (Figure 4.13 and Figure 4.14)
    - Min: 50,000
    - Max: 50,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.

These four series of automated games were run with and without the above changes to the source code, a total of 8 series of automated games. We can refer to Figure 4.5 for the specific metadata values used.

As before, the two series that ”played the same game” should be compared in order to ensure that the correct conclusions are drawn.

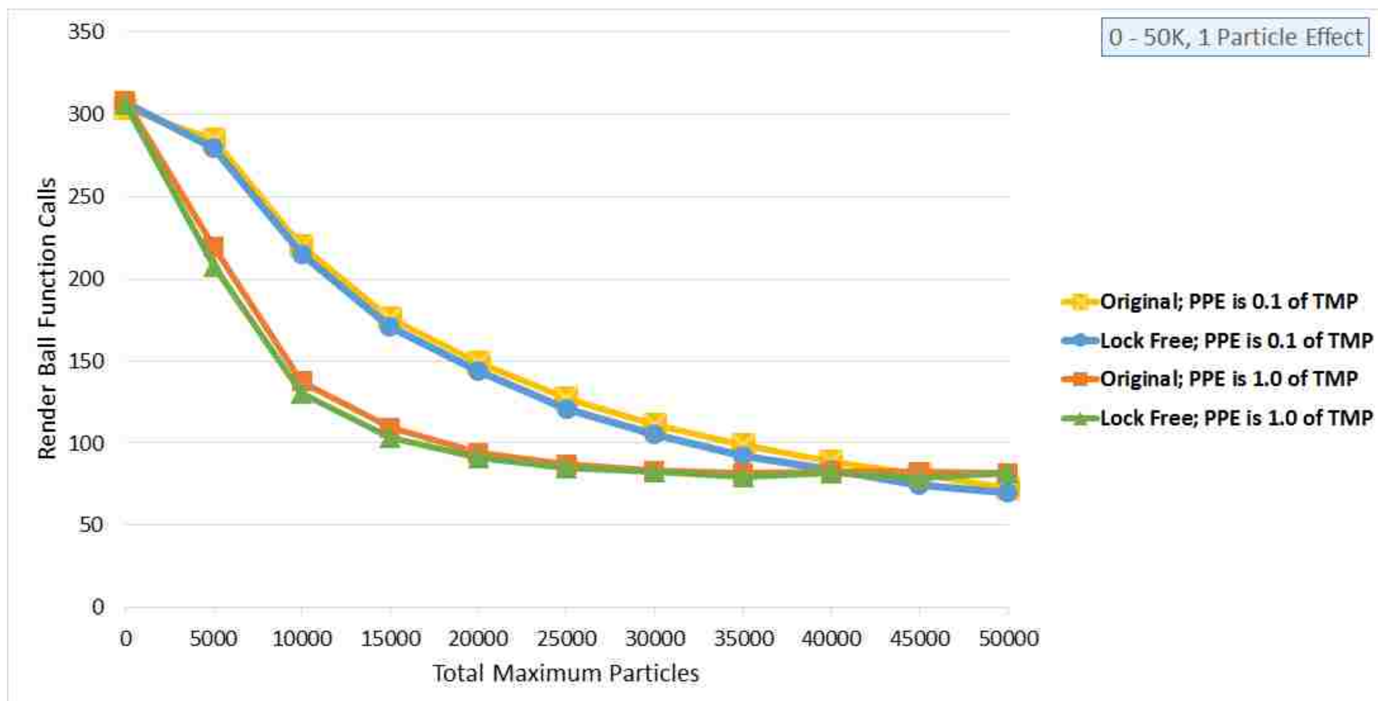

Figure 4.11 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows that changing `Concurrent Particle Data` to use a lock free data structure does not impact the number of times that the ball is rendered. The multiple-burst and single-burst (0.1 and 1.0 TMP, respectively) scenarios for the Lock Free data series are almost the same as the ones for the data series without any changes to the source code (i.e. the Original data series).

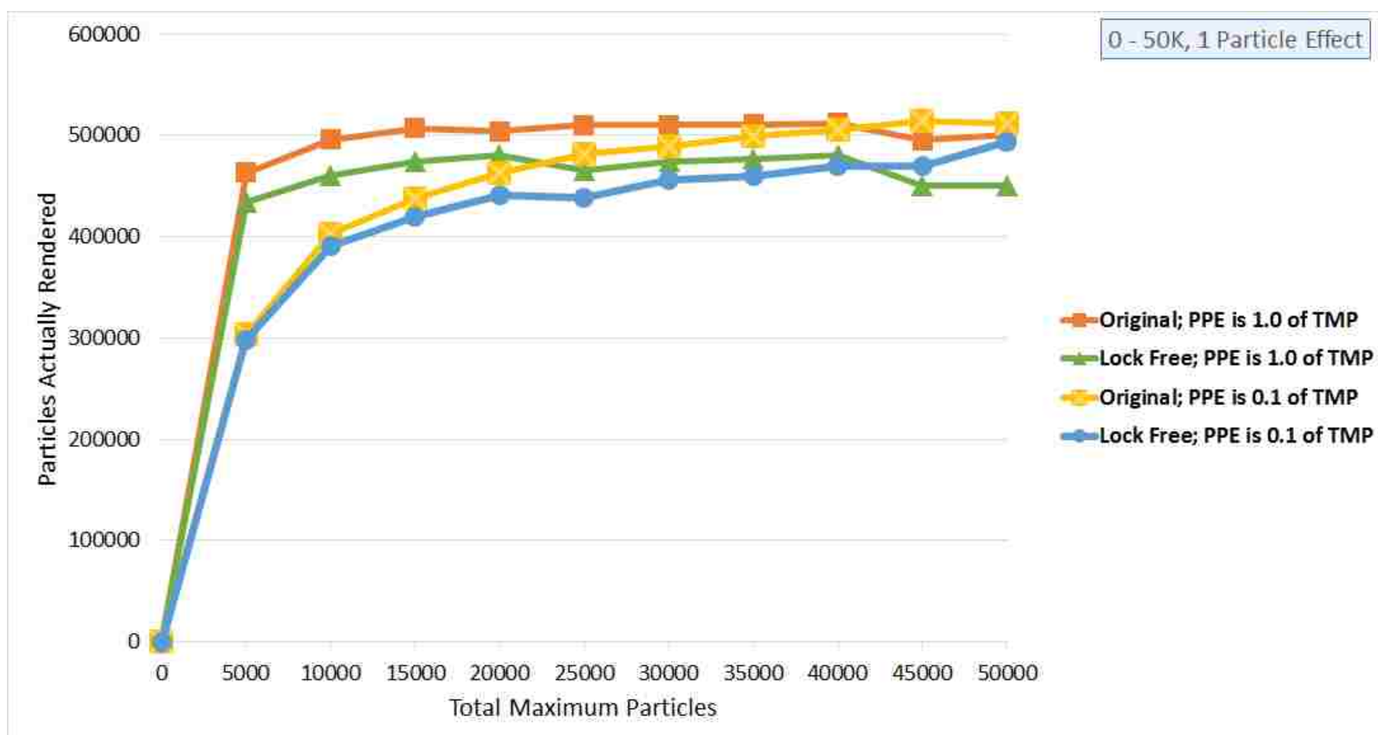

Figure 4.12 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. This shows that changing `Concurrent Particle Data` to use a lock free data structure does not positively impact the number of times that particles are rendered. The multiple-burst and single-burst (0.1 and 1.0 TMP, respectively) scenarios for the Lock Free data series are lower (which means worse performance) than the ones for the Original data series.

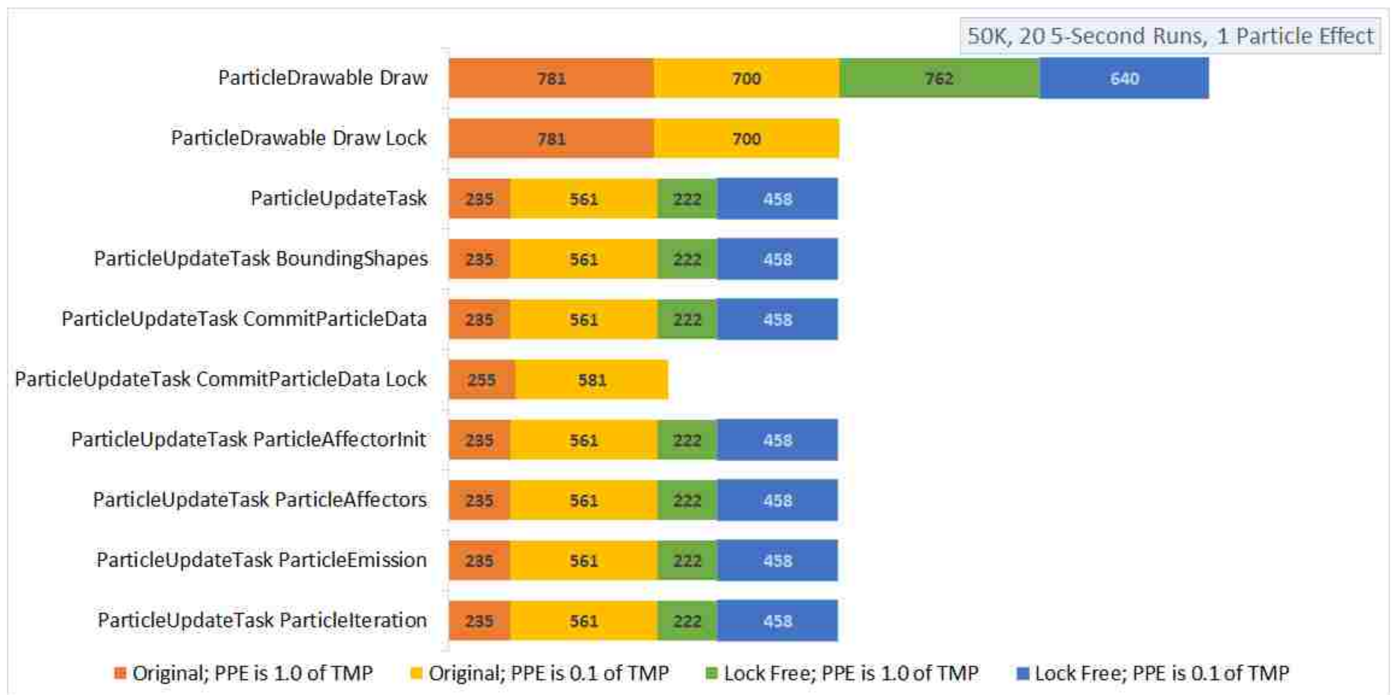

Figure 4.13 A graph showing the number of section calls for various sections of code for a single particle effect. This shows that, between the Lock Free and Original data series, the drawing functions had approximately the same number of section calls. The Original data series for the particle update functions also had approximately the same number of function calls; however, the Lock Free data series for 0.1 TMP had a little less section calls than the Original.

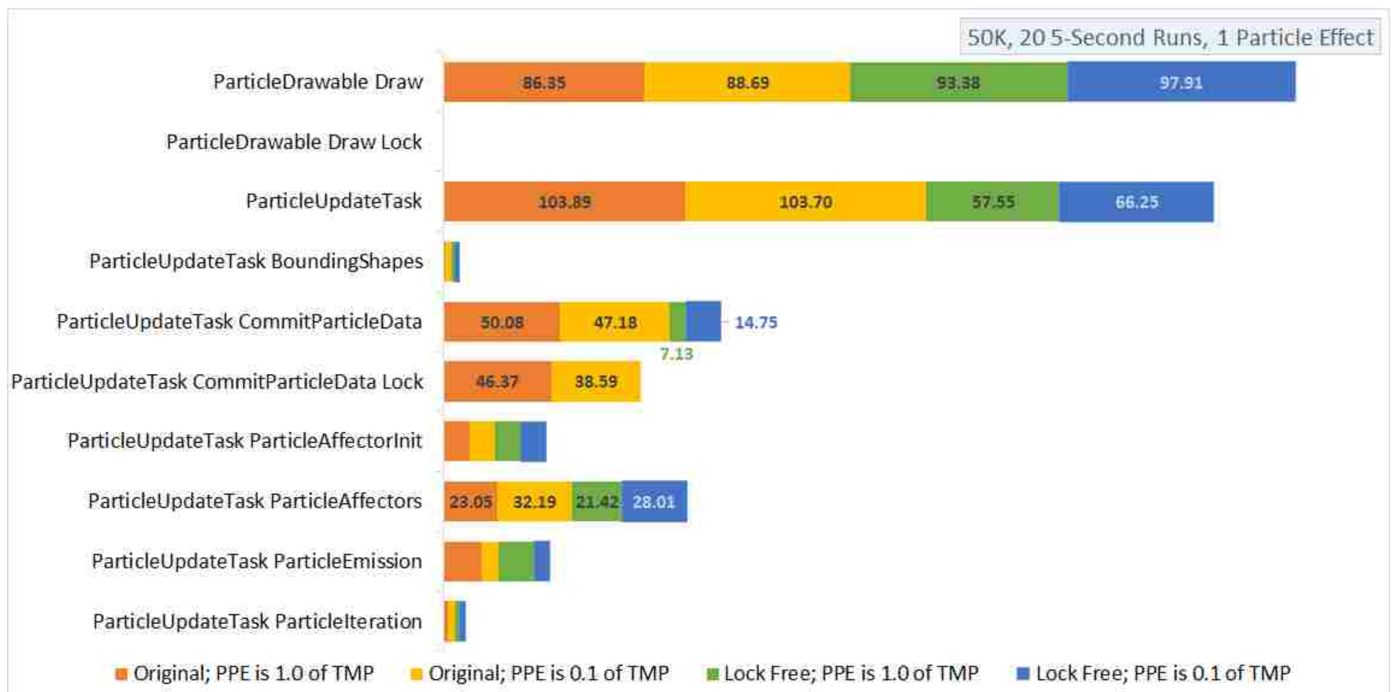

Figure 4.14 A graph showing the time spent in various sections of code for a single particle effect. `CommitParticleData` for the Lock Free data series, as expected, experienced a performance boost. For 0.1 TMP, the function went down from 47 seconds to 15. For 1.0 TMP, it went down from 50 seconds to 7. This was, respectively, about an 68% and 86% increase in performance.

4.3.3 Discussion

The *particles actually rendered* and *render ball function calls* figures, similar to the first case study, do not show significant improvement from the original. However, the Lock Free data series in the *particles actually rendered* figure shows slightly lower numbers than the original.

As predicted, the timing metric figures show that `CommitParticleData` experienced a healthy performance boost. For 0.1 TMP, the function went down from 47 seconds to 15. For 1.0 TMP, it went down from 50 seconds to 7. This was, respectively, about an 68% and 86% boost![5]A larger boost was seen in the latter scenario due to the fact that the there were more active particles earlier in the game(s), and more contention was expected since the draw function needed to render more particles. This blocked `CommitParticleData` for longer, and minimizing that increased wait time resulted in better run times.

Although these were excellent initial results for the the Lock Free case study, there were two problems:

- The particle metric figures did not improve.
- The timing metric were only favorable for a single particle effect.

The first problem was an issue because these optimizations should have aided in rendering more particles since updating them took less time. As for the second problem, the next section will aide in understanding how the two case studies behaved quite differently when the games were run with multiple particle effects instead of just one.

[5]See Figure 4.24 in Section 4.4 for a summary table of these timed sections.

4.4 Unexpected Results with Multiple Particle Effects

attribute	10 PE, 0-5K	10 PE, 5K
are particles looping	TRUE	TRUE
num particle effects	10	10
min particles	0	5K
max particles	5K	5K
particles step	500	500
is total max particles changing	TRUE	TRUE
is particles per emission changing	TRUE	TRUE
particles per emission	0	0
total maximum particles	0	0
particles per emission step	{50, 500}	{50, 500}
total maximum particles step	500	500
total num runs	5	20
duration per run	5	5

Figure 4.15: The metadata used for the 8 series of automated games for the unexpected results from the case studies.

In order to exhibit the unexpected results for the Multiple Mutex and Lock Free case studies, 8 automated games were run per case study as we did before. This time, however, the particles were distributed across 10 particle effects. For example, the scenario that had a particle range of 0 to 50K would now have 0 to 5K per particle effect. For clarification, please refer to Figure 4.15.

Sections 4.4.1 and 4.4.2 both show the results of running the same games as before but with 10 particle effects attached to the ball instead of just 1.

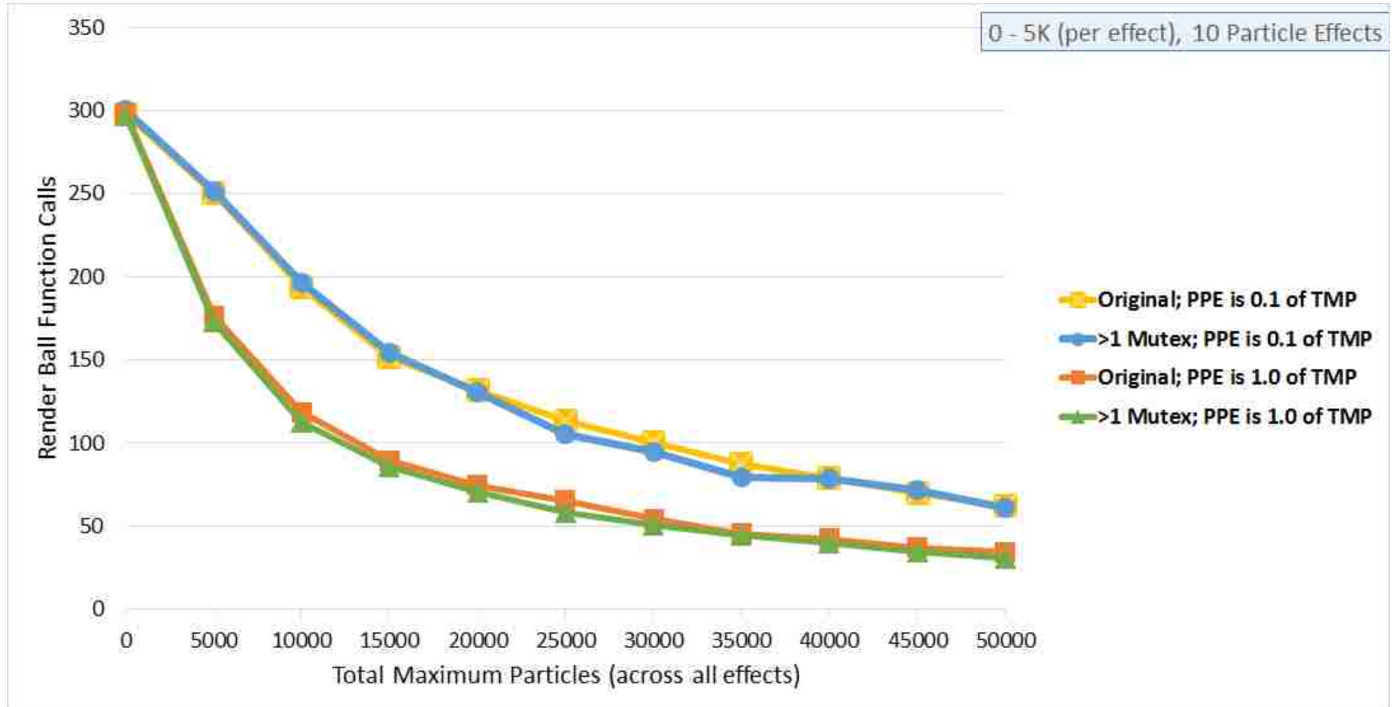

Figure 4.16 A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. The two data series, > 1 Mutex and Original, show no stark differences; their *render ball function calls* values are nearly the same for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

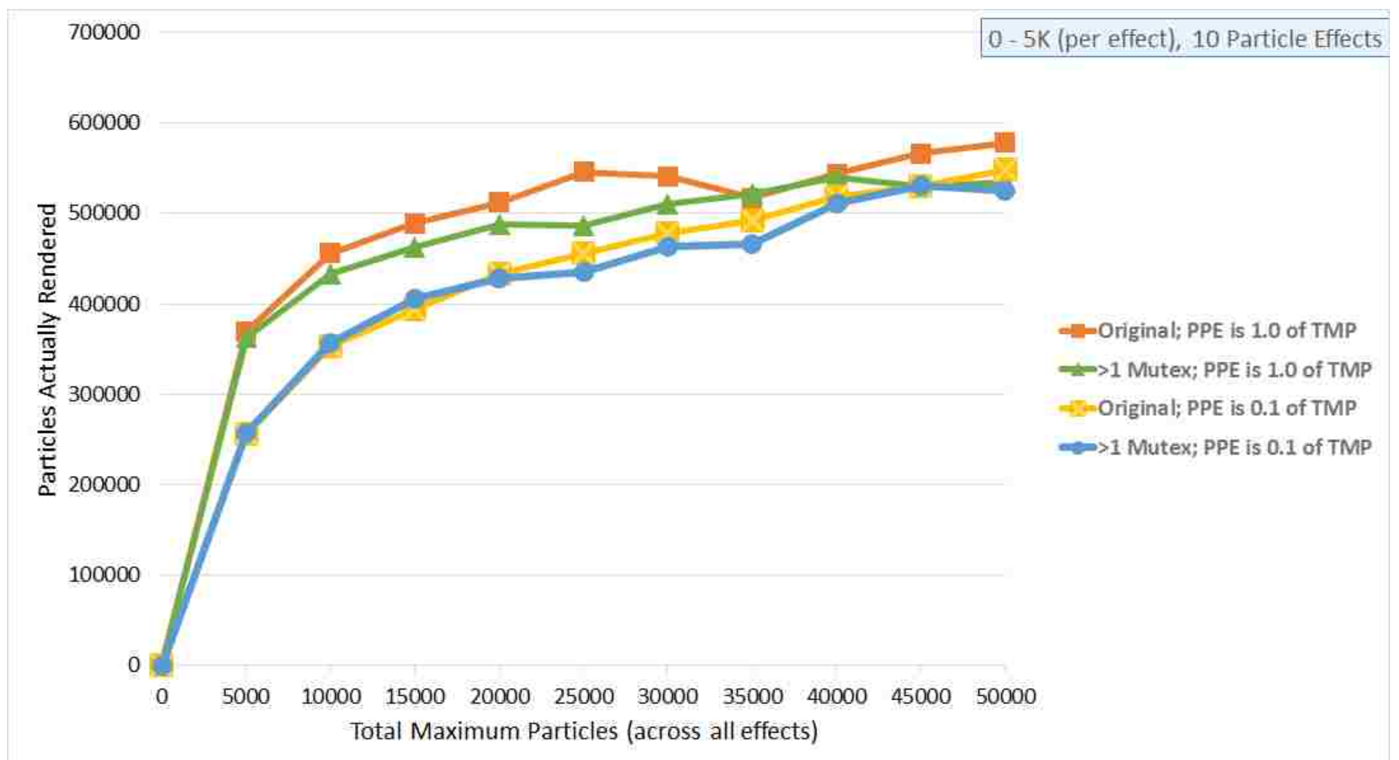

Figure 4.17 A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. The two data series, > 1 Mutex and Original, show no stark differences; their *particles actually rendered* values are nearly the same for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

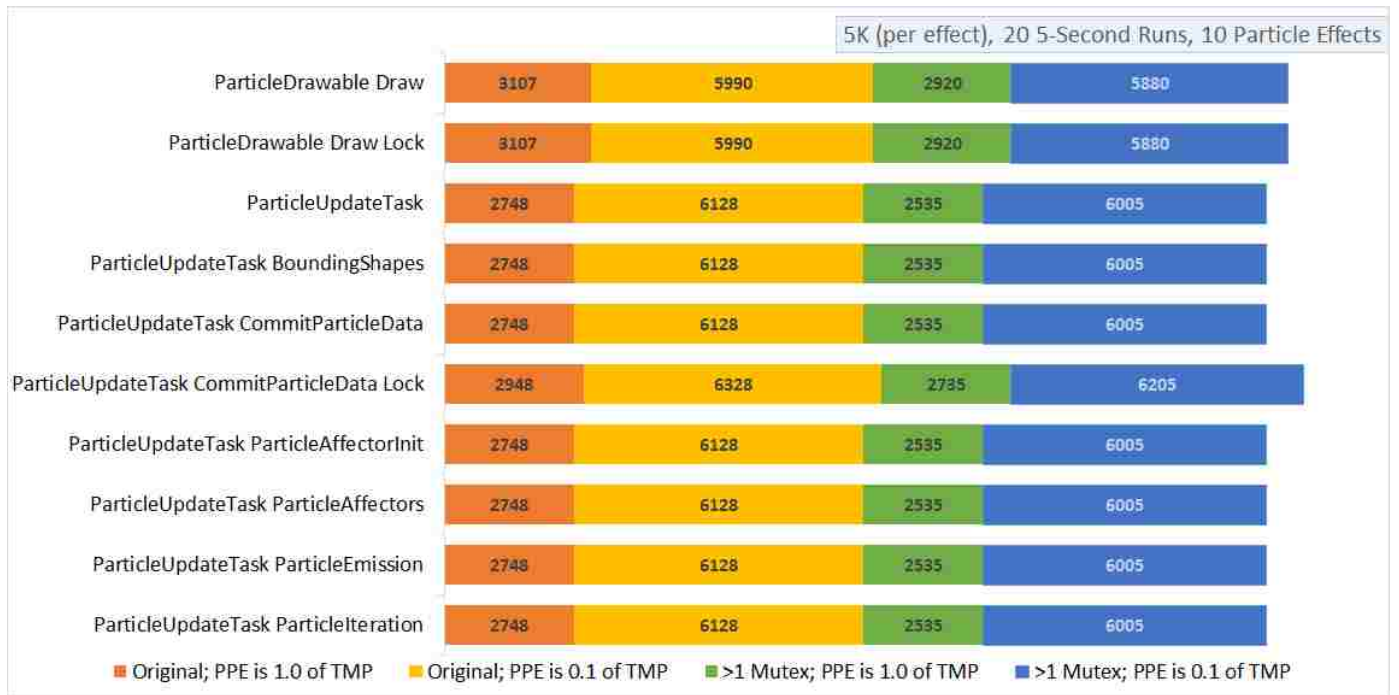

Figure 4.18 A graph showing the number of section calls for various sections of code for ten particle effects. The two data series, > 1 Mutex and Original, show no stark differences; the section call values are the same for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

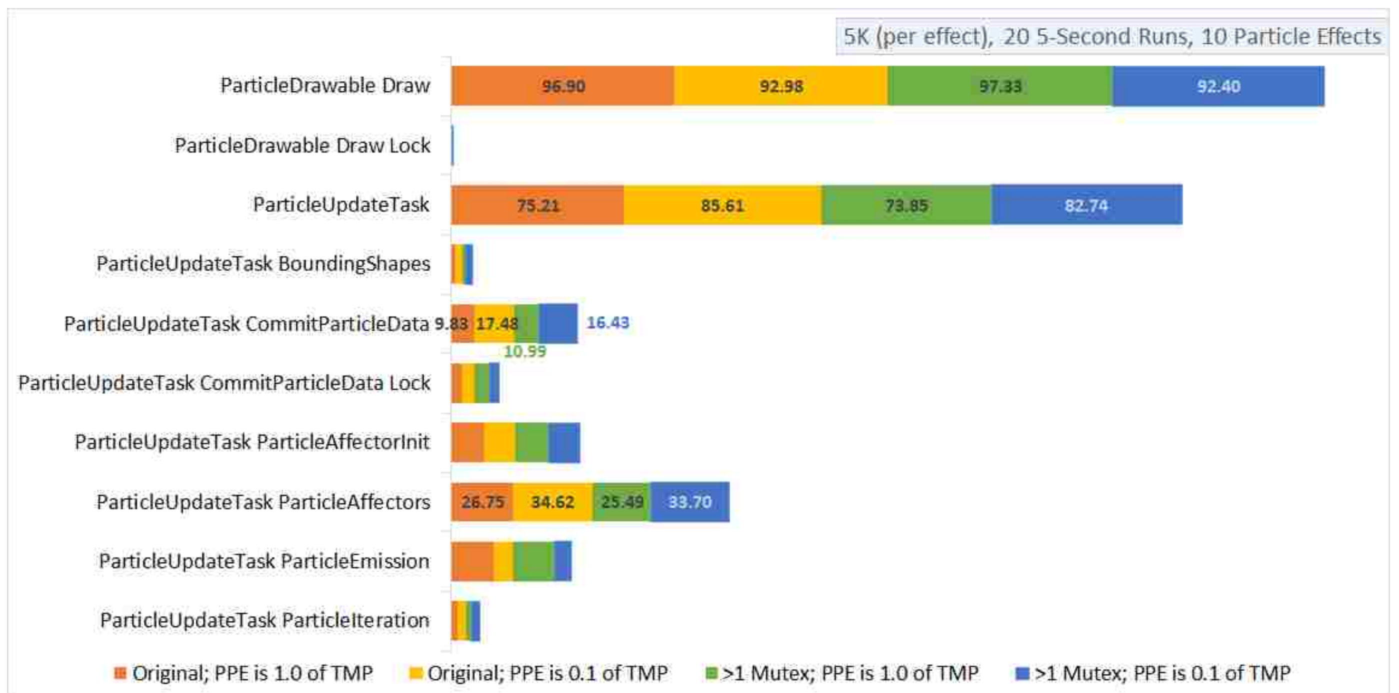

Figure 4.19 A graph showing the time spent in various sections of code for ten particle effects. Unlike in the single particle effect results, this shows not only no significant improvement to the `CommitParticleData` function, but it also shows that there was little contention to begin with in the Original data series.

4.4.2 Lock Free Results

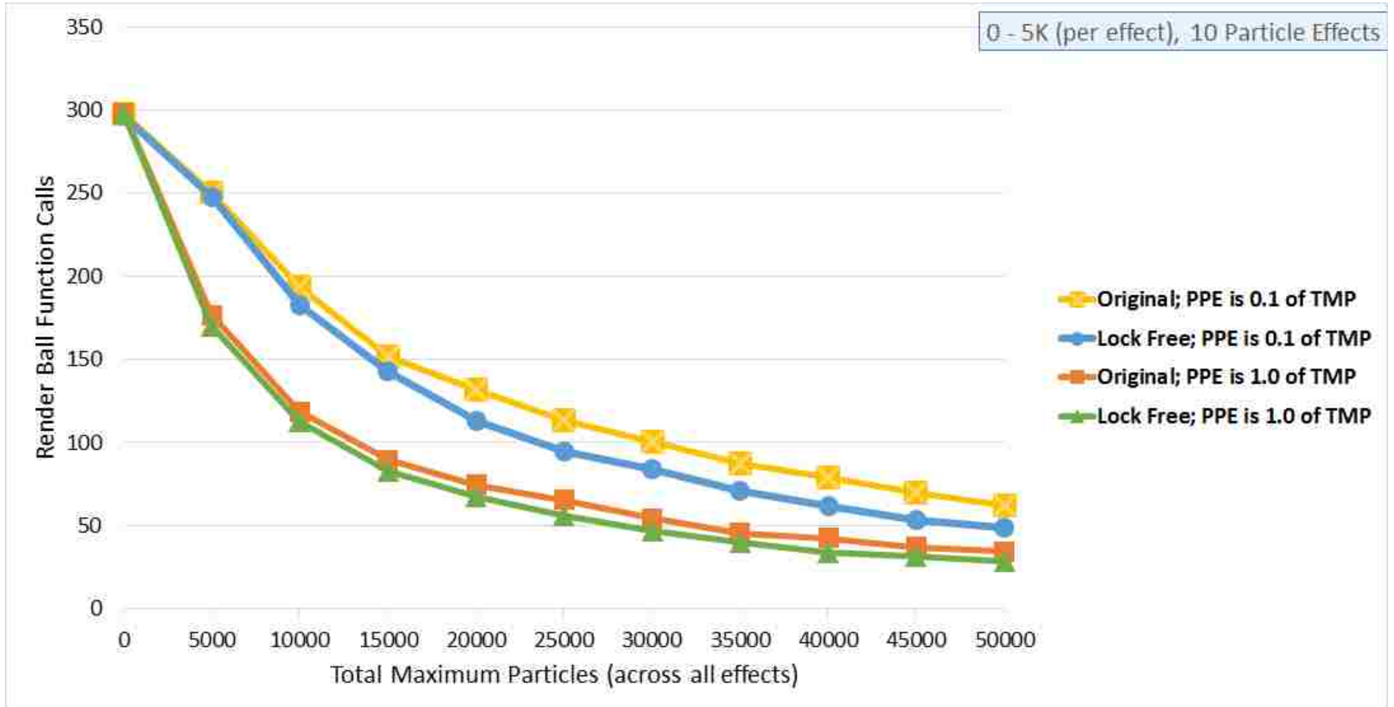

Figure 4.20 A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. Although they are very similar, the Lock Free data series exhibits slightly lower *render ball function calls* values than the Original data series.

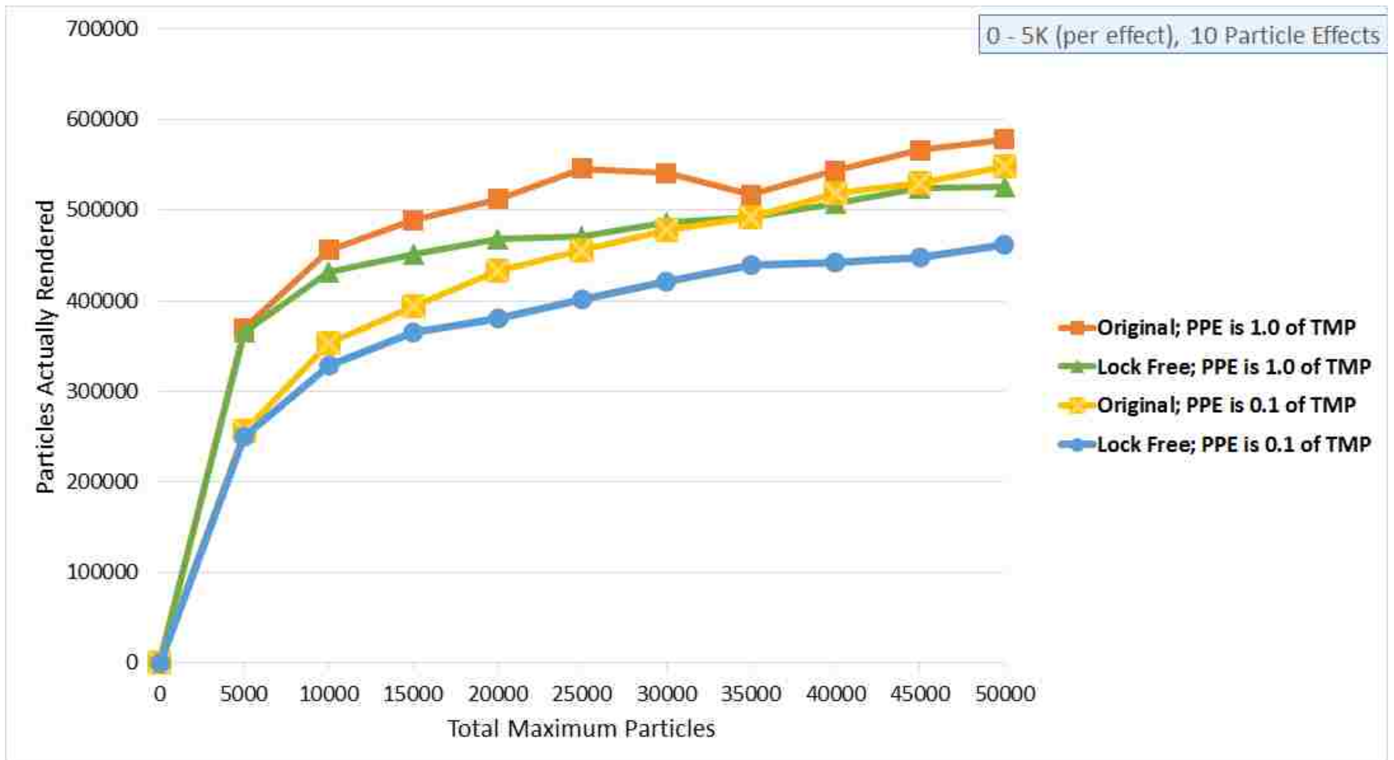

Figure 4.21 A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. Similar to the single particle effect results, the multiple-burst and single-burst (0.1 and 1.0 TMP, respectively) scenarios for the Lock Free data series are lower than the ones for the Original data series.

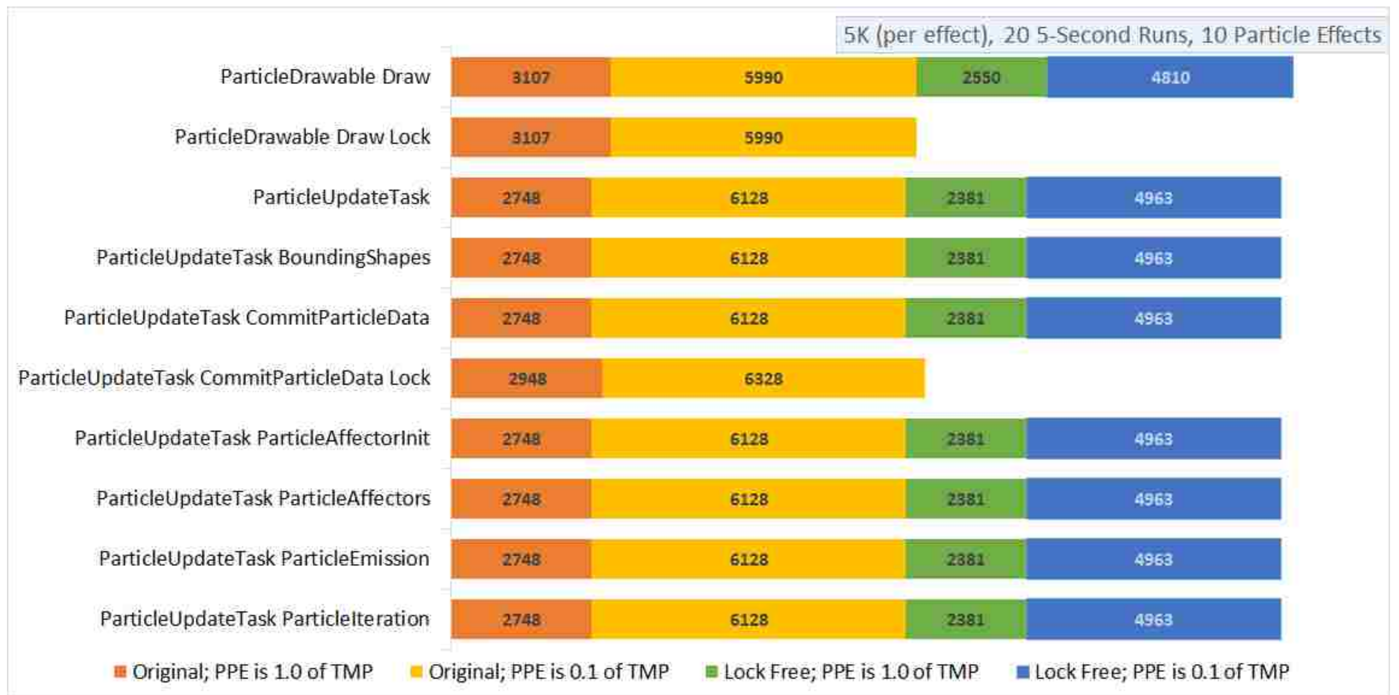

Figure 4.22 A graph showing the number of section calls for various sections of code for ten particle effects. The two data series, Lock Free and Original, show no stark differences. However, the Original data series seems to have a slightly higher section call count than the Lock Free data series for both the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

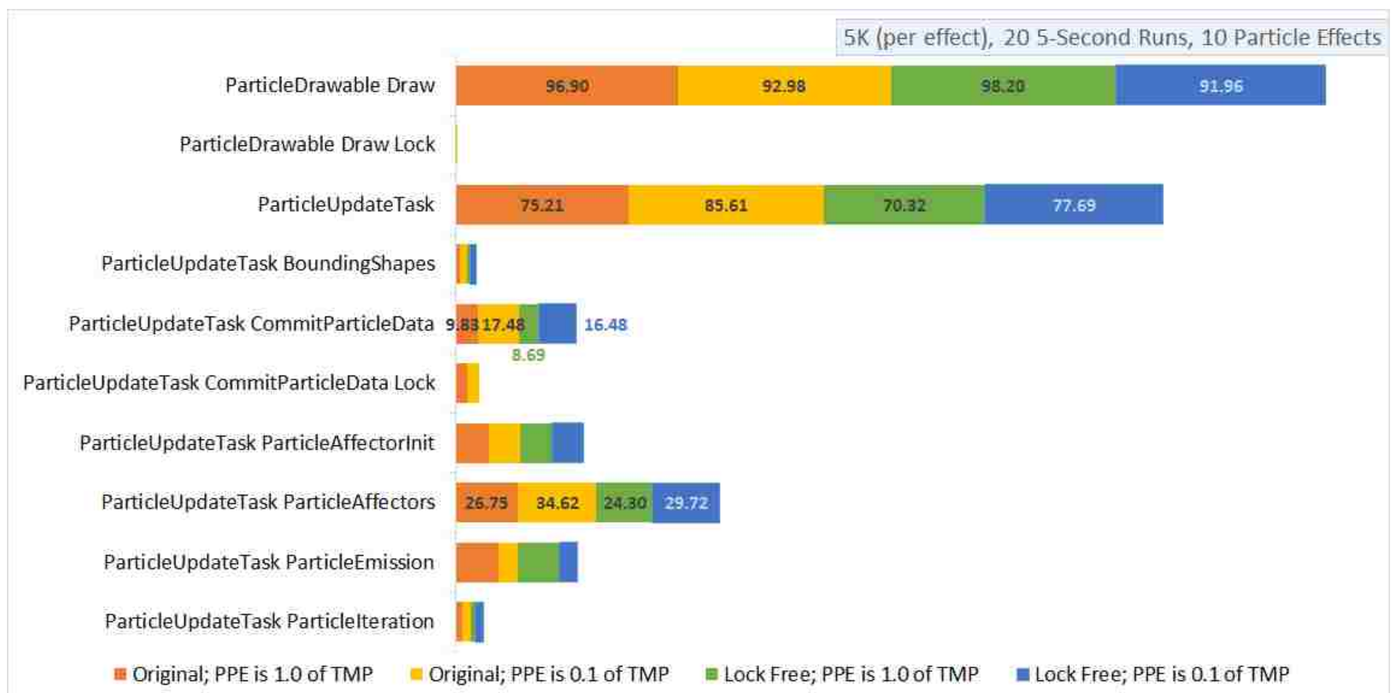

Figure 4.23 A graph showing the time spent in various sections of code for ten particle effects. Unlike in the single particle effect results, this shows not only no significant improvement to the `CommitParticleData` function, but it also shows that there was little contention to begin with in the Original data series.

4.4.3 Discussion

The most interesting (and, perhaps, frustrating) part of the results shown in sections 4.4.1 and 4.4.2 was that all of the data series that "played the same game" looked essentially the same. The ball was rendered nearly as many times for the four scenarios for both case studies, and the `particles actually rendered` were also similar for both as well. However, just like in the Lock Free case study results, there were slightly fewer particles rendered than in the original. To try to understand what happened, let us examine figures 4.24 and 4.25. In particular, let us focus on the "1.0 TMP" scenario with the Original and Lock Free implementations.

Within the 1.0 TMP Original results for a single particle effect, about 50 of the 104[6] seconds spent in the function `ParticleUpdateTask` are dedicated to calling `CommitParticleData`. Further, about 46 of these 50 seconds in `CommitParticleData` were spent waiting for a lock to release. Within the 1.0 TMP Lock Free results for a single particle effect, about 7 of the 58 seconds spent in the function `ParticleUpdateTask` were spent calling `CommitParticleData`.

Within the 1.0 TMP Original results for ten particle effects, about 9 of the 75 seconds spent in the function `ParticleUpdateTask` were dedicated to calling `CommitParticleData`. Further, about 5 of these 9 seconds in `CommitParticleData` were spent waiting for a lock to release. Within the 1.0 TMP Lock Free results for a single particle effect, about 9 of the 75 seconds spent in the function `ParticleUpdateTask` were spent calling `CommitParticleData`.

For a single effect, it can be seen that the Lock Free implementation did an excellent job in reducing contention for a large data structure between a background thread writing to it and a main thread reading from it. For ten effects, the Lock Free implementation did little to reduce the contention because there was no contention to begin with. This

[6] It may not make sense why there are greater than 100 seconds here since we ran 20 runs for 5 seconds, but it was mostly due to the timer starting only when the ball was activated after the entire scene had been created.

Lock Free, 0.1 TMP, 1 effect w/ 50K each, 20 5-second runs			Lock Free, 1.0 TMP, 1 effect w/ 50K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	97.91	640	ParticleDrawable Draw	93.38	762
ParticleDrawable Draw Lock	0.00	0	ParticleDrawable Draw Lock	0.00	0
ParticleUpdateTask	66.25	458	ParticleUpdateTask	57.55	222
ParticleUpdateTask CommitParticleData	14.75	458	ParticleUpdateTask CommitParticleData	7.13	222
ParticleUpdateTask CommitParticleData Lock	0.00	0	ParticleUpdateTask CommitParticleData Lock	0.00	0

Original, 0.1 TMP, 1 effect w/ 50K each, 20 5-second runs			Original, 1.0 TMP, 1 effect w/ 50K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	88.69	700	ParticleDrawable Draw	86.35	781
ParticleDrawable Draw Lock	0.05	700	ParticleDrawable Draw Lock	0.10	781
ParticleUpdateTask	103.70	561	ParticleUpdateTask	103.89	235
ParticleUpdateTask CommitParticleData	47.18	561	ParticleUpdateTask CommitParticleData	50.08	235
ParticleUpdateTask CommitParticleData Lock	38.59	581	ParticleUpdateTask CommitParticleData Lock	46.37	255

>1 Mutex, 0.1 TMP, 1 effect w/ 50K each, 20 5-second runs			>1 Mutex, 1.0 TMP, 1 effect w/ 50K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	92.61	776	ParticleDrawable Draw	89.14	771
ParticleDrawable Draw Lock	0.03	776	ParticleDrawable Draw Lock	0.03	771
ParticleUpdateTask	62.36	358	ParticleUpdateTask	65.69	145
ParticleUpdateTask CommitParticleData	24.31	358	ParticleUpdateTask CommitParticleData	31.10	145
ParticleUpdateTask CommitParticleData ParticleLock	18.89	378	ParticleUpdateTask CommitParticleData ParticleLock	29.05	165

Figure 4.24 A summary of the timed sections for the unexpected results for the Lock Free and > 1 Mutex case studies for a single particle effect. This highlights that, for a single particle effect, the Original Data series clearly experienced performance-hindering contention in CommitParticleData. The changes made in the > 1 Mutex and Lock Free data series significantly reduced this contention.

Lock Free, 0.1 TMP, 10 effects w/ 5K each, 20 5-second runs			Lock Free, 1.0 TMP, 10 effects w/ 5K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	91.96	4810	ParticleDrawable Draw	98.20	2550
ParticleDrawable Draw Lock	0.00	0	ParticleDrawable Draw Lock	0.00	0
ParticleUpdateTask	77.69	4963	ParticleUpdateTask	70.32	2381
ParticleUpdateTask CommitParticleData	16.48	4963	ParticleUpdateTask CommitParticleData	8.69	2381
ParticleUpdateTask CommitParticleData Lock	0.00	0	ParticleUpdateTask CommitParticleData Lock	0.00	0

Original, 0.1 TMP, 10 effects w/ 5K each, 20 5-second runs			Original, 1.0 TMP, 10 effects w/ 5K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	92.98	5990	ParticleDrawable Draw	96.90	3107
ParticleDrawable Draw Lock	0.27	5990	ParticleDrawable Draw Lock	0.12	3107
ParticleUpdateTask	85.61	6128	ParticleUpdateTask	75.21	2748
ParticleUpdateTask CommitParticleData	17.48	6128	ParticleUpdateTask CommitParticleData	9.83	2748
ParticleUpdateTask CommitParticleData Lock	5.00	6328	ParticleUpdateTask CommitParticleData Lock	5.02	2948

>1 Mutex, 0.1 TMP, 10 effects w/ 5K each, 20 5-second runs			>1 Mutex, 1.0 TMP, 10 effects w/ 5K each, 20 5-second runs		
timed section name	time in seconds	num section calls	timed section name	time in seconds	num section calls
ParticleDrawable Draw	92.40	5880	ParticleDrawable Draw	97.33	2920
ParticleDrawable Draw Lock	0.21	5880	ParticleDrawable Draw Lock	0.11	2920
ParticleUpdateTask	82.74	6005	ParticleUpdateTask	73.85	2535
ParticleUpdateTask CommitParticleData	16.43	6005	ParticleUpdateTask CommitParticleData	10.99	2535
ParticleUpdateTask CommitParticleData ParticleLock	4.28	6205	ParticleUpdateTask CommitParticleData ParticleLock	6.47	2735

Figure 4.25 A summary of the timed sections for the unexpected results for the Lock Free and > 1 Mutex case studies for ten particle effects. This highlights that, for many particle effects, the Original Data series did not experience a great deal of contention in CommitParticleData. Since there was no contention to begin with, the changes made in the $> 1 Mutex$ and Lock Free data series did not make a significant impact.

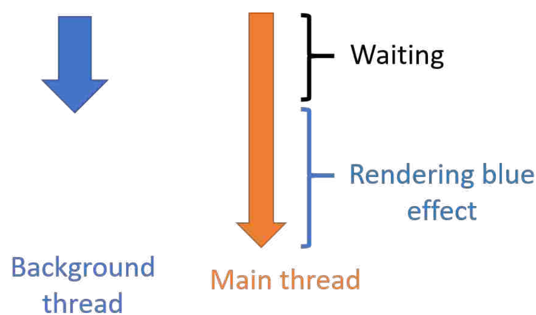

Figure 4.26 The lifetime of a single particle effect in relation to its scheduled background thread and the main thread.

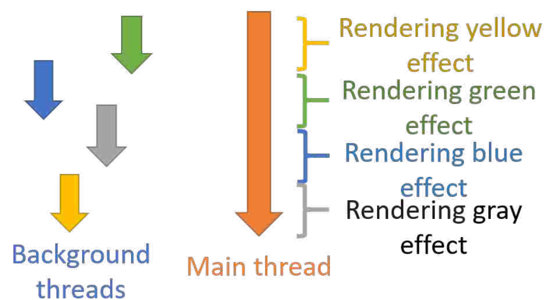

Figure 4.27 The lifetime of a many particle effects in relation to their scheduled background threads and the main thread.

can be observed by the fact that the results from the Original implementation with ten effects showed that, compared with a single effect, an insignificant amount of time was spent waiting for a lock to release.

This behavior can be better understood if we consider the lifetime of particle effects in two scenarios: one with many effects, and one with a single effect. The one with many particle effects would have its particles distributed across many `Particle Effect Components` and, thus, many `Particle Drawables`. The `Renderer` running on the main thread, then, renders each `Particle Effect Component` separately. When the main thread requests to lock any given effect's particle array mutex, it has a reduced chance of waiting for it. This is because each `Particle Effect Component` has its own background thread and each one is only responsible for a fraction of the particles. For example, using 10 effects would mean approximately a 1/10th chance of contending with any of the particle effects. The one with a single particle effect, on the other hand, has one `Particle Effect Component` with a large array of particles that it has to manage on its own. It can only schedule one background thread at a time and is responsible for updating every single particle. When the `Renderer` running on the main thread requests to lock the effect's particle array mutex, it has a greater chance of waiting for it.

The unexpected results of these case studies demonstrated the differences between man-

aging contention for a single, large data structure and for many pieces or ”shards” of a fragmented data structure. The former was likely to have contention if there was writing involved between two threads, and a solution that utilized ”lock free” atomic read-modify-write operations could significantly reduce contention. The latter, however, was unlikely to have contention between two threads, and so the developer could get away with naively using mutexes to synchronize their data.

CHAPTER 5 CHILLISOURCE PARTICLE EFFECT COMPONENT OBSERVATION STUDIES

5.1 Overview

I performed the optimization case studies in the previous chapter to understand the contention between the main thread that renders particles and the background thread that updates them. This chapter explores various other aspects of *`Particle Effect Components`*. I examined three aspects of the particle effect: (1) the importance of the `ConcurrentParticleData` object, (2) the relationship between ChilliSource's *`Task Scheduler`* and *`Particle Effect Components`*, and (3) the perplexing scenarios in which particle effects do not render particles.

5.2 Copying Data with ConcurrentParticleData

5.2.1 Overview

As discussed in prior chapters, the `ConcurrentParticleData` object contains copies of primitives, objects, and data structures needed by the *`Particle Drawable`* to render the particles. It also handles all of the locking necessary in order to synchronize the background and main threads. This section addresses what occurred when `ConcurrentParticleData` was not used and the member variables within *`Particle Effect Component`* were instead directly used by the *`Particle Drawable`*.

The results showed that using the `ConcurrentParticleData` object for a single particle effect shifts the heavy contention to itself instead of the *`Particle Drawable`*. In

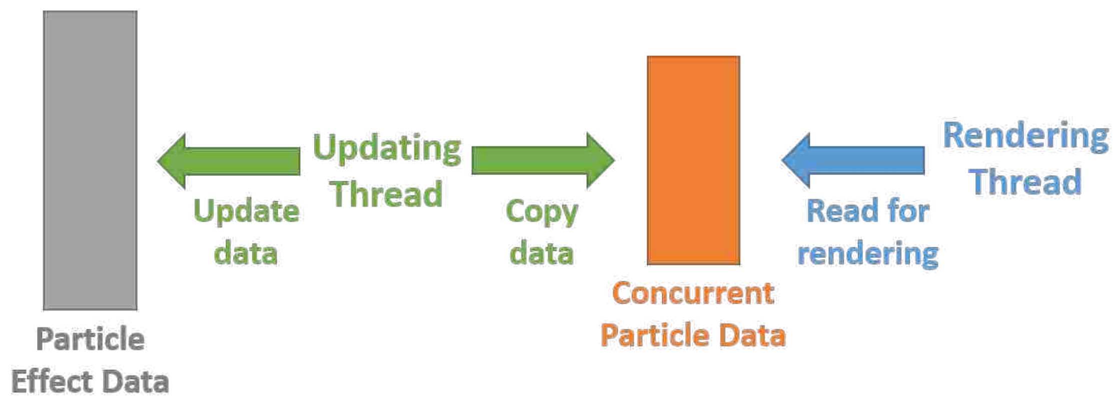

Figure 5.1 Demonstrates how the updating and rendering threads use the *ConcurrentParticleData* object to communicate information about the particle effect. After the updating thread updates the particle data directly from the particle effect, it copies a subset of the particle data that is necessary for rendering to the *ConcurrentParticleData* instance. The rendering thread reads from the *ConcurrentParticleData* object and not directly from the particle effect itself.

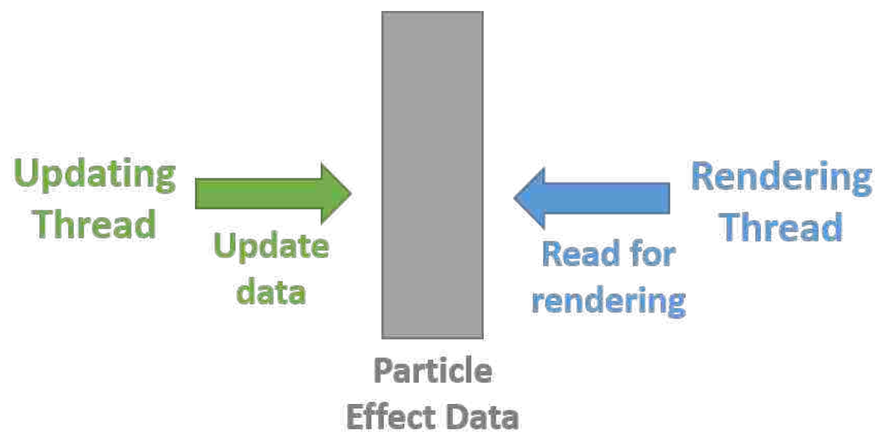

Figure 5.2 Demonstrates how, during this observation study, updating and rendering threads worked directly with the particle effect data; the *ConcurrentParticleData* object was completely removed.

other words, if the `ConcurrentParticleData` object was not used during a game with a single particle effect attached to the ball, then the *Particle Drawable* would take on the contention that was observed earlier on this paper. For 10 particle effects, however, there was not any contention to begin with so using the `ConcurrentParticleData` object made little difference.

5.2.2 Code

There were a considerable number of changes that were needed to remove ConcurrentParticleData from the *Particle Effect Component* and *Particle Drawable* classes:

- Modified the *Particle Drawable* to use pointers from the *Particle Effect Component* and removed the ConcurrentParticleData member variable.
- Added variables that ConcurrentParticleData was keeping track of.
- Added functions that ConcurrentParticleData provided. (Which primarily included accessor and locking/unlocking functions)
- Modified ParticleUpdateTask to update member variables from the *Particle Effect Component*.
- Modified the *Particle Effect Component* and *Particle Drawable* to utilize locks and mutexes in order to ensure thread safety.

The majority of the above changes are described in Code Listings 17 and 18, but it is also necessary to expand on some points.

In addition to keeping track of information needed for rendering, ConcurrentParticleData has two booleans that the *Particle Effect Component* uses for scheduling updates: hasActiveParticles and isUpdating. These booleans describe whether or not there are any particles active in the effect and if ParticleUpdateTask is running, respectively. If hasActiveParticles is false when the component is updating on the main thread, then the component stops[1]. If isUpdating is false when the component is updating on the main thread, then the component schedules a ParticleUpdateTask. In order to replace this functionality from ConcurrentParticleData, I added these booleans to *Particle Effect Component* as atomic booleans passed by reference to the ParticleUpdateTask function.[2]

[1]The particle effect will only completely stop if looping is false as well.

[2]These booleans are not shown in the pseudocode for simplicity, but their behavior is simple; hasActiveParticles will switch to false if it did not update any particles in ParticleUpdateTask, and isUpdating will switch to false at the end of ParticleUpdateTask.

Since the `Particle Effect Component` uses the bounding information objects on the main thread and the `Drawable` does not, I created two mutexes for the bounding information and the rest of the rendering data.

As shown by Code Listing 17, the `ParticleUpdateTask` is locked by a member variable mutex. It also passes references to new particle indicies (i.e. `copiedAtts.newParticleIndices`) and to the bounding information (i.e. `copiedAtts.AABB` and `copiedAtts.Sphere`).

Code Listing 17: Pseudocode describing the important changes needed to remove the `ConcurrentParticleData` instance from the `Particle Effect Component`.

```
ParticleEffectComponent::OnUpdate()
{
  // Get a struct containing copies (and pointers) to pass to the task
  copiedAtts = CopyUpdateAttributes();

  // Schedule a task to update the particle in the background
  ApplicationTaskScheduler.ScheduleTask
  ({
    // NEW -> Lock the mutex for the whole task
    using lock(this.mutex)
    {
      ParticleUpdateTask(copiedAtts);
    }
  });
}

ParticleUpdateTask(copiedAtts)
{
  // Update particles if they are active
  for(particle in copiedAtts.particleArray)
  {
    if(particle.isActive)
      particle.UpdateValues();
  }

  // Apply affectors
  for(affector in copiedAtts.particleAffectors)
  {
    affector.AffectParticles();
  }

  // Try to emit
  newParticleIndices = copiedAtts.particleEmitter.EmitParticles();

  // NEW -> Append the new indices to the member new indices array
  // (instead of passing it to CommitParticleData)
  copiedAtts.newParticleIndices.append( newParticleIndices.copy() );

  // NEW -> Update member bounding shapes; lock the bounding shapes mutex
```

```
40      // (instead of passing it to CommitParticleData)
41      using lock(copiedAtts.boundingMutex)
42      {
43        copiedAtts.AABB, copiedAtts.Sphere = CalculateBoundingShapes();
44      }
45    }
```

Code Listing 18 shows that the `Particle Drawable` did not change significantly. The most notable change is that the `Draw` function is now locked by `Particle Effect Component`'s mutex within the `Render` function. Additionally, the particles array is not passed to the `DrawParticles` function. Instead, the array is passed into the drawable when it is created by the `Particle Effect Component`.

Code Listing 18: Pseudocode describing the important changes needed to remove the `ConcurrentParticleData` instance from the `Particle Drawable`.

```
1   ParticleEffectComponent::Render(camera)
2   {
3     // Use the Particle Drawable to draw particles to the screen
4     if(this.playbackState is playing or this.playbackState is stopping)
5     {
6       // NEW -> Lock the member mutex to keep particles and new particle indices safe
7       using lock(this.mutex)
8       {
9         this.drawable.Draw(camera, this.TakeNewParticleIndices());
10      }
11    }
12  }
13
14  // NEW -> newParticleIndicies is passed to the Draw function
15  Drawable::Draw(camera, newParticleIndices)
16  {
17    //  NEW -> No locking; it's locked above in Render.
18
19    // Activate the newly emitted particles in the static billboard
20    for(particleIndex in newParticleIndices)
21    {
22      ActivateParticleInBillboard(particleIndex);
23    }
24
25    // Iterate through all of the particles and draw each one
26    // NEW -> The particles array is a member variable of the drawable
27    DrawParticles(camera);
28  }
```

5.2.3 Results

The metrics shown here are the result of running a series of automated games in two scenarios:

1. Scenario used for particle metrics (Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6)
    - Min: 0
    - Max: 50,000
    - Step: 5,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
    - This scenario ran with 1 and 10 particle effects
2. Scenario used for timing metrics (Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.10)
    - Min: 50,000
    - Max: 50,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
    - This scenario ran with 1 and 10 particle effects

These 8 series of automated games were run with and without the above changes to the source code for a total of 16 series of automated games.

5.2.4 Discussion

The *particles actually rendered* (Figure 5.3) and *render ball function calls* (Figure 5.4) charts for 1 particle effect show something interesting. The No Object data series (the data series without the `ConcurrentParticleData` object) rendered the ball fewer times

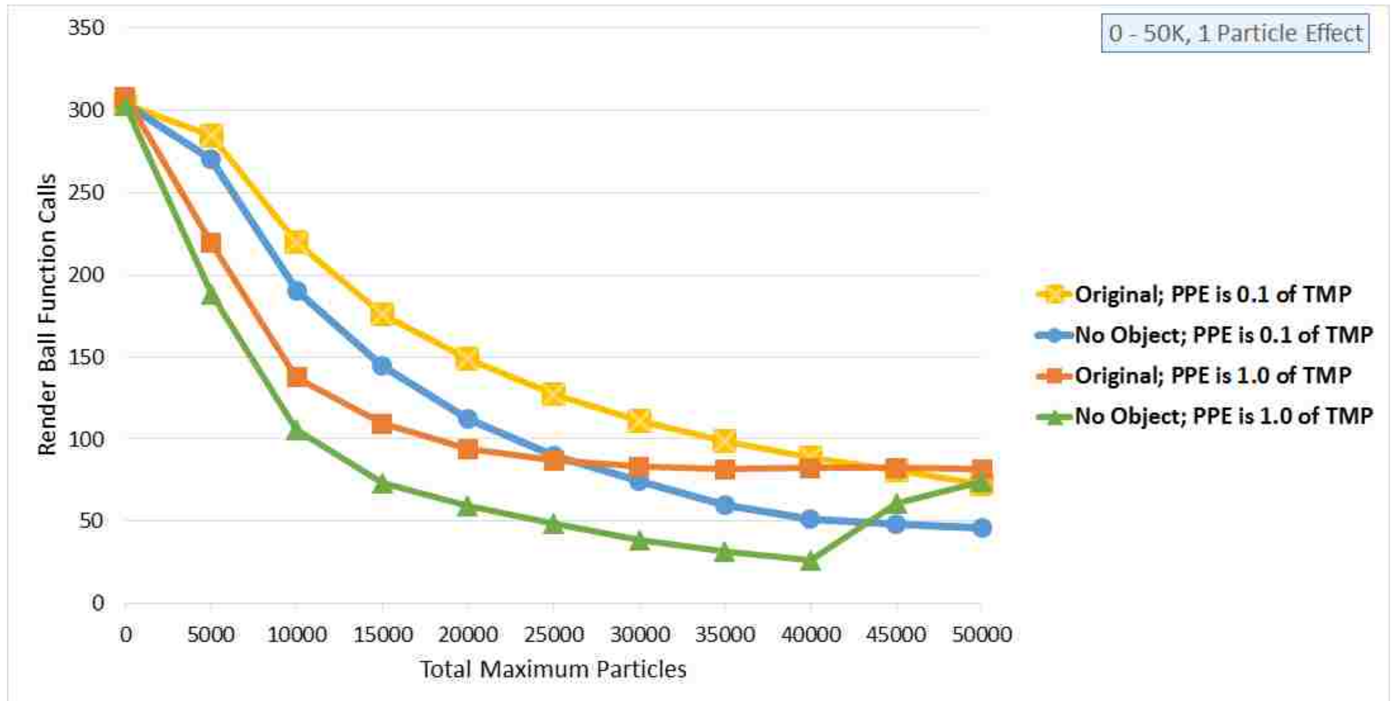

Figure 5.3 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows that the No Object data series exhibits worse performance than the Original data series. The *render ball function calls* values from the Original data series are consistently higher than the No Object data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

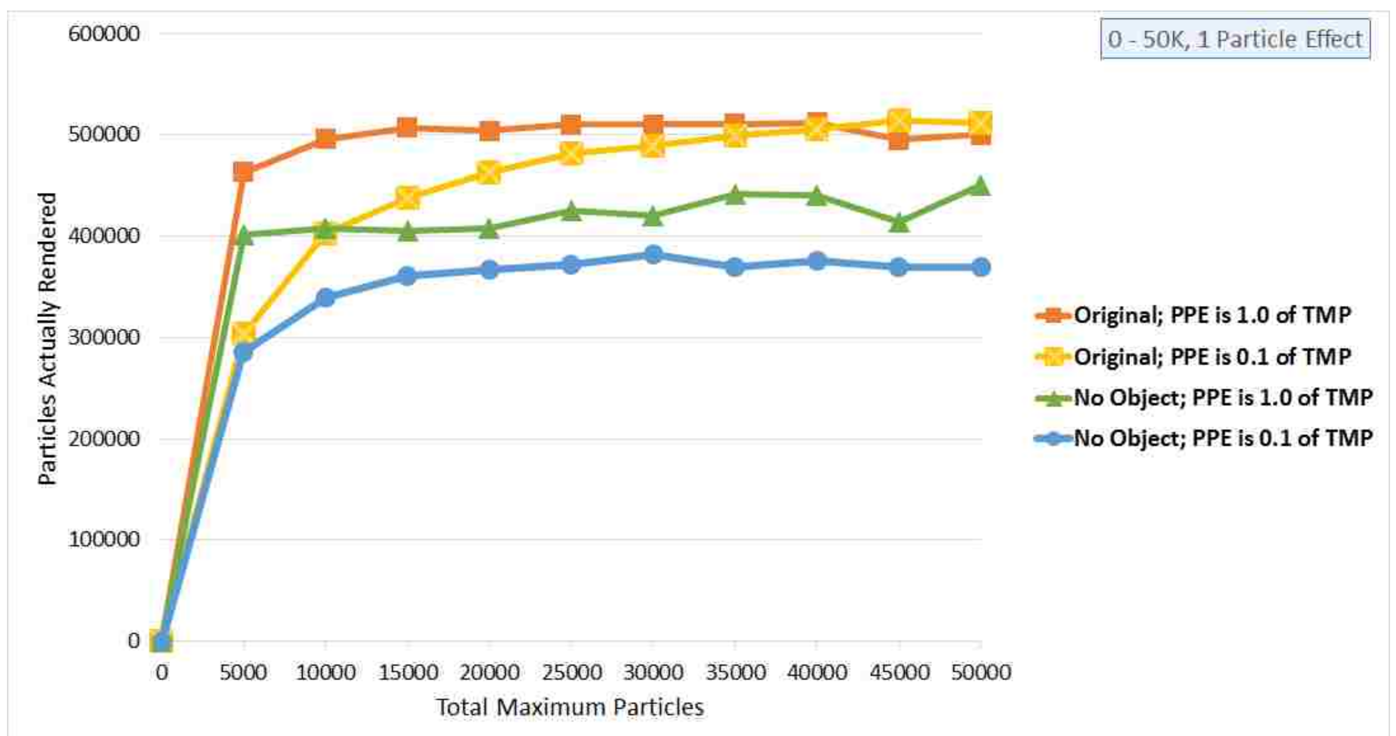

Figure 5.4 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. Similarly to Figure 5.3, this shows that the Original data series performs better (i.e. renders more particles) than the No Object data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

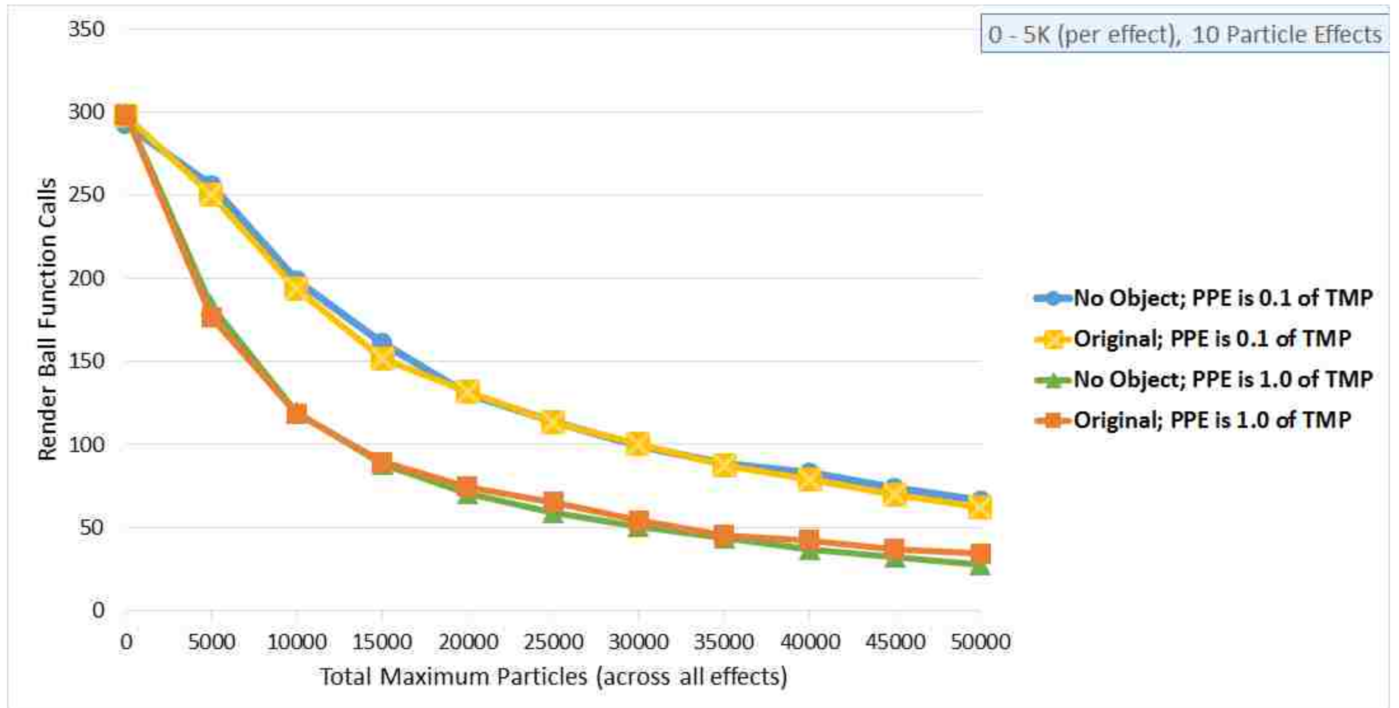

Figure 5.5 A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. This shows little to no difference between the *render ball function calls* values from the Original and No Object data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

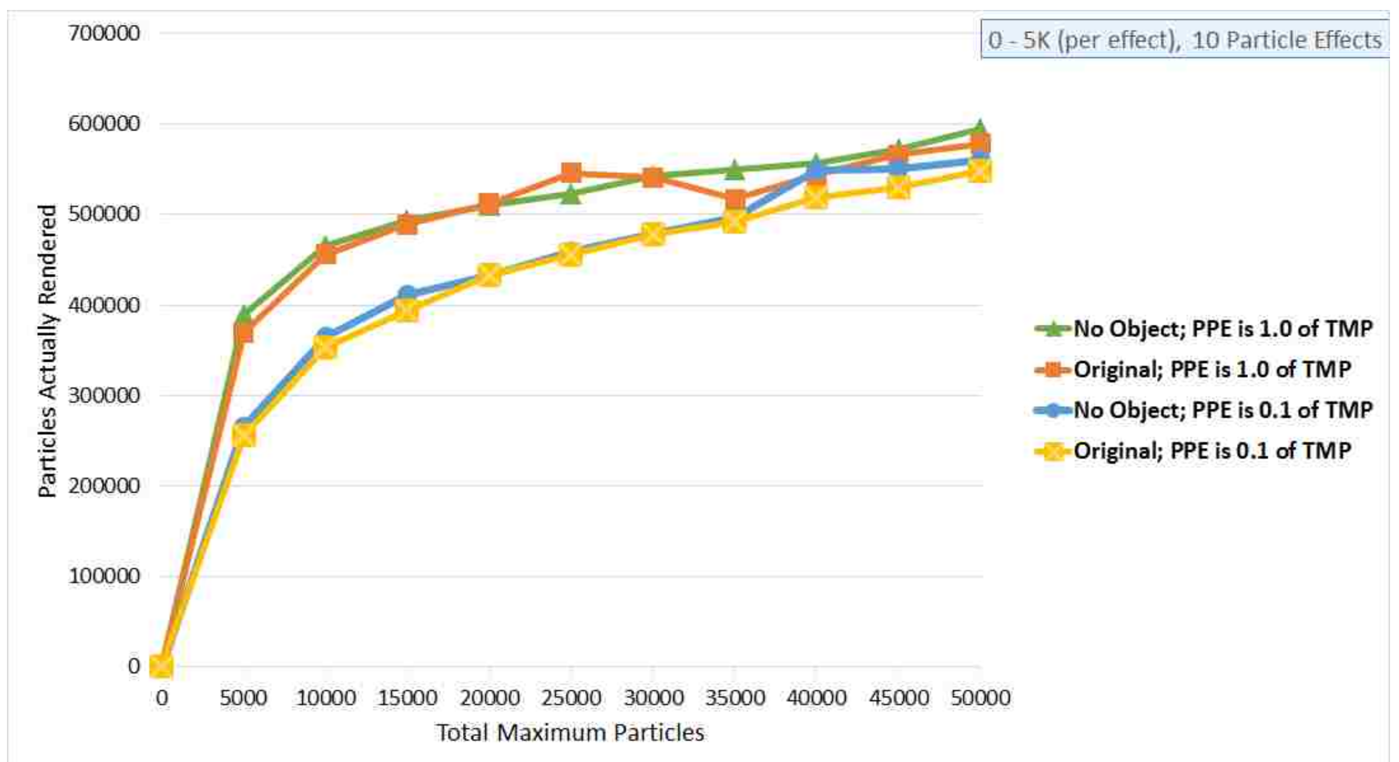

Figure 5.6 A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. Like in Figure 5.5, the *particles actually rendered* values from the Original and No Object data series are very similar.

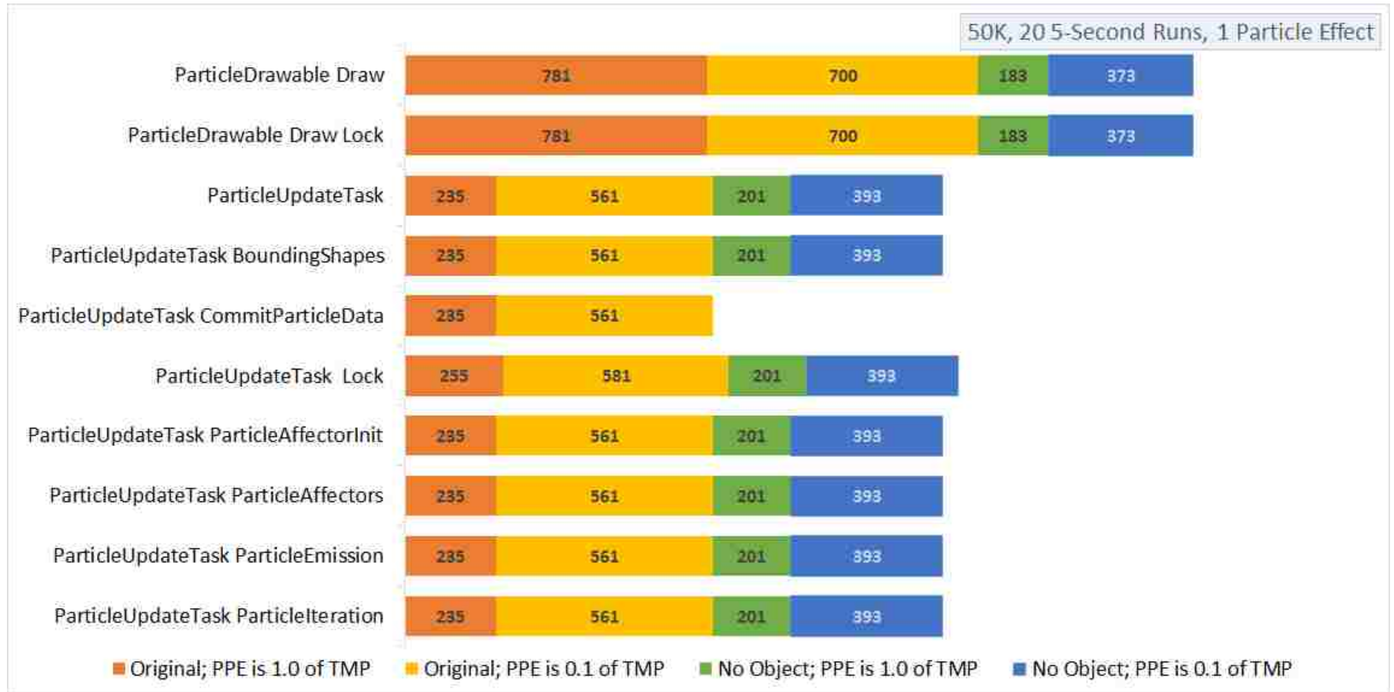

Figure 5.7 A graph showing the number of section calls for various sections of code for a single particle effect. This most notably shows that the section calls for the Original data series had significantly higher values in the drawing functions then the ones in the No Object data series.

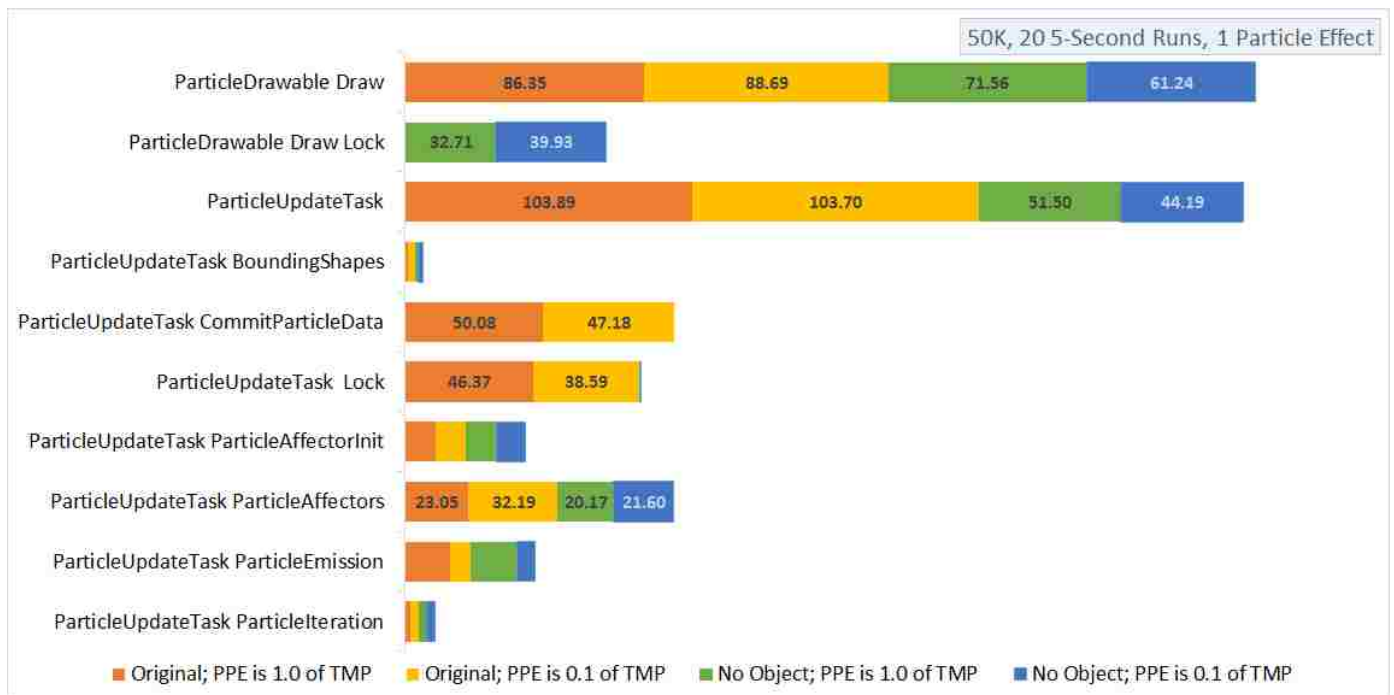

Figure 5.8 A graph showing the time spent in various sections of code for a single particle effect. This illustrates how the contention was moved from `CommitParticleData` to `Draw` when the `ConcurrentParticleData` object was not used (i.e. the No Object data series).

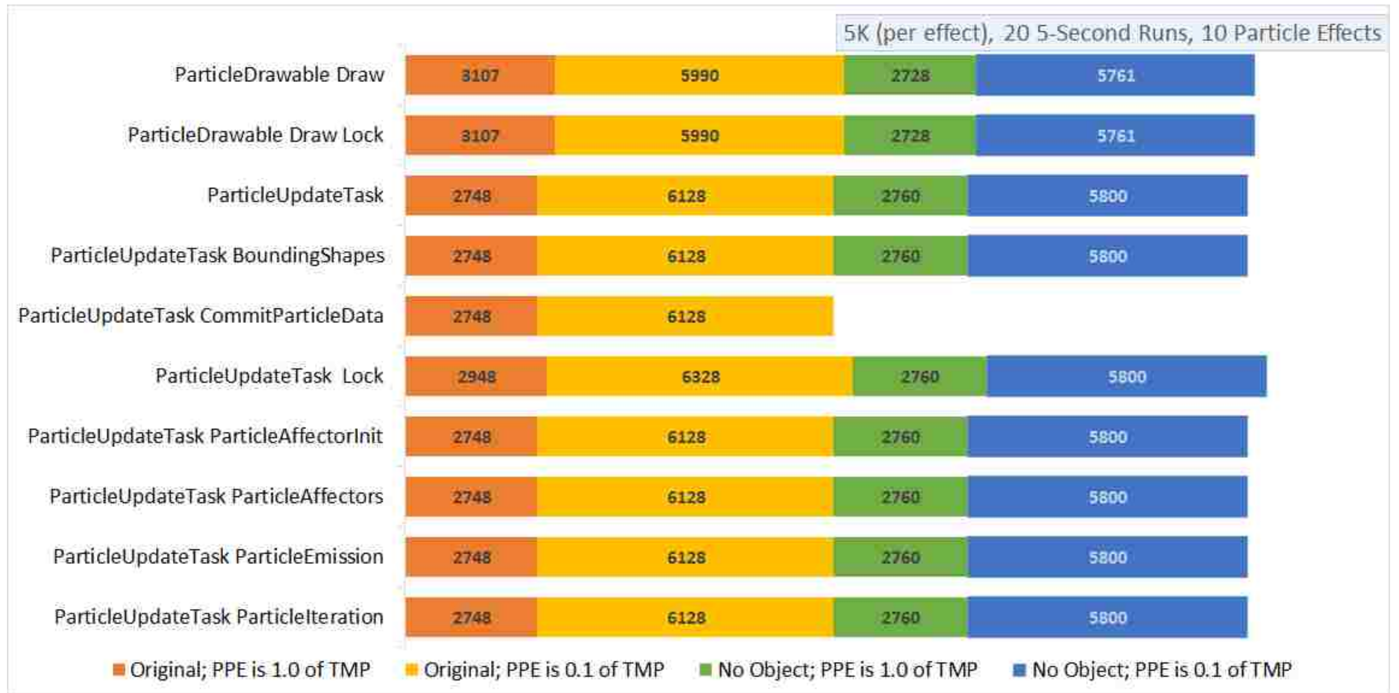

Figure 5.9 A graph showing the number of section calls for various sections of code for ten particle effects. This shows that removing the `ConcurrentParticleData` object for many particles did not affect the number of section calls; the Original and No Object data series both have similar values for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

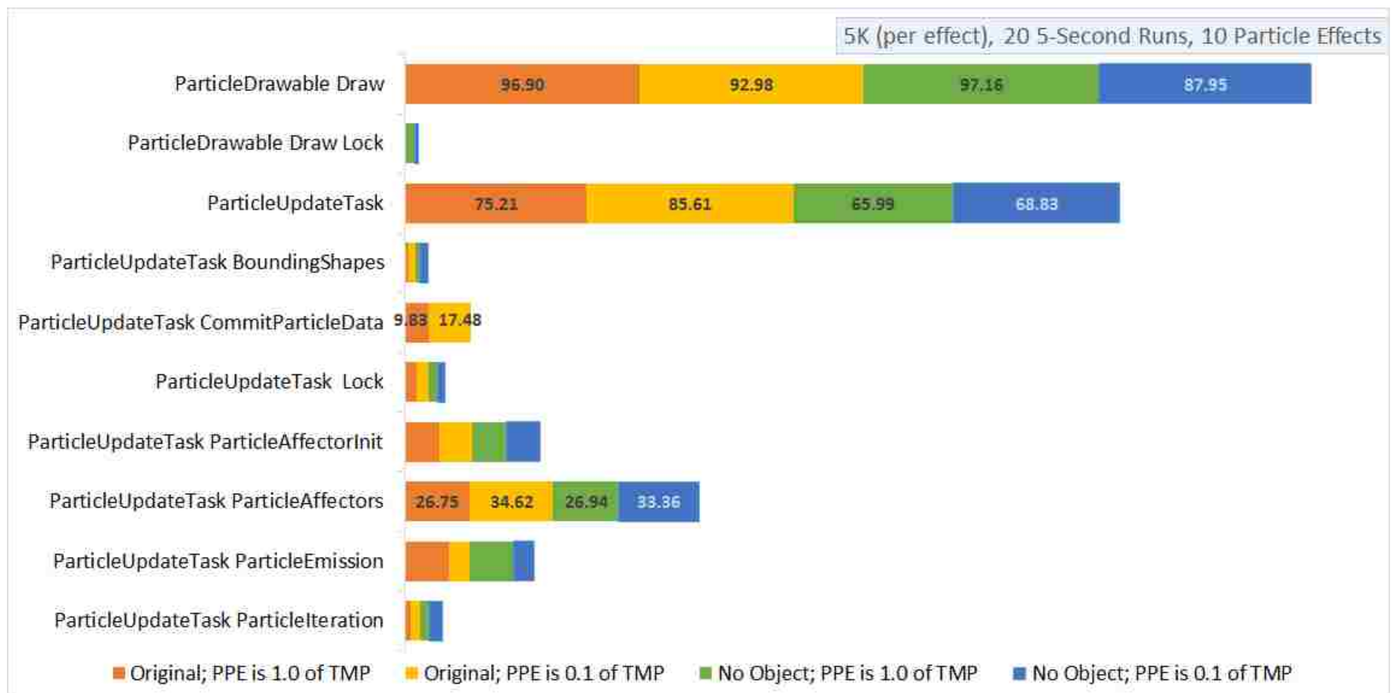

Figure 5.10 A graph showing the time spent in various sections of code for ten particle effects. Like in Figure 5.9, this shows that the Original and No Object data series both have similar run times for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

than the original, and it also rendered its particles fewer times. An examination of the timing metrics for 1 particle effect (Figure 5.8) shows why this was occurring. Instead of the background thread waiting for the mutex to release in `ParticleUpdateTask`, the contention had moved to the *`Particle Drawable`* in the `Draw` function.

If we further examine Figure 5.8, it shows that the `Draw` function in the No Object data series spends less time rendering than the Original data series. Although a lower run-time usually means that the engine is performing better, it is not true in this case. I did not modify the amount of time that it takes to render all of the effect's particles and, for the single-burst (PPE is 1.0 of TMP) particle effect, only 72 - 33 = 39 seconds of meaningful work was done since the rendering thread spent about 33 seconds waiting for a lock to release. Since it spent less time in the `Draw` function, it rendered fewer particles.

In other words, the *`Concurrent Particle Data`* object improves performance by increasing the number of particles that are drawn to the screen which results in a smoother visual experience for the end user.

The charts for 10 particle effects, however, do not demonstrate this performance benefit. Both the Original and No Object data series show nearly identical results. This is because there was less overall contention when the particles were distributed across effects, as discovered in the previous chapter. Although the benefits of using a `ConcurrentParticleData` object was not as significant when using many particle effects, it is appealing from a software engineering perspective. Encapsulating the thread synchronization to a single class is clearer and more maintainable, and that will be useful if the developer desires to refactor the *`ParticleEffectComponent`*.

5.3 Multithreading and Task Scheduling

5.3.1 The ChilliSource Task Scheduler

As mentioned earlier, `ParticleUpdateTask` is scheduled as a background thread by ChilliSource's *`Task Scheduler`*. The *`Task Scheduler`* can schedule a variety of task

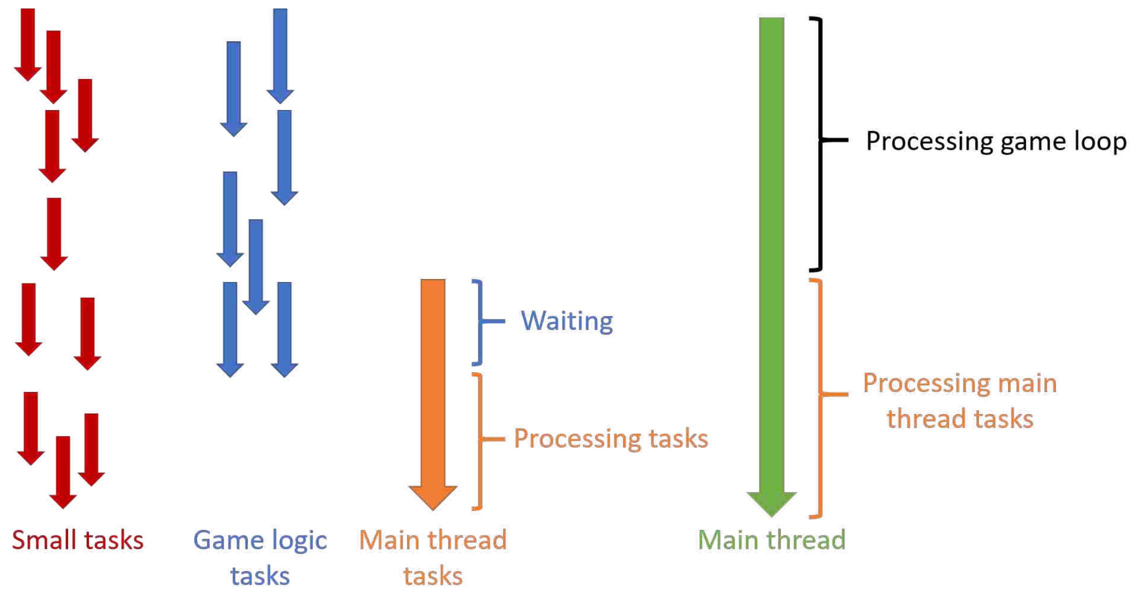

Figure 5.11 Illustrates the lifetime of *small tasks*, *game logic tasks*, and *main thread tasks* in relation to the main thread. The important concept to understand from this figure is how *small tasks* are ran whenever regardless of what the main thread is doing and the *main thread tasks* are bound by the main thread. The *game logic tasks* can also be technically ran as the main thread is processing the game loop, but it is special in that it is guaranteed to be done by the end of the current frame.

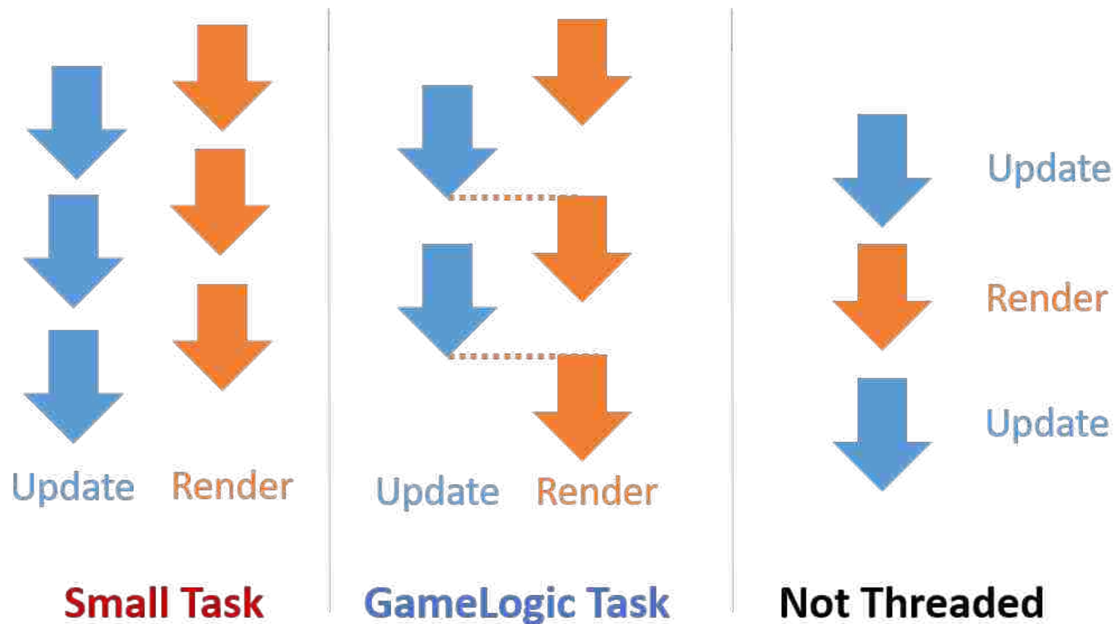

Figure 5.12 Illustrates the relationship between the updating and rendering processes when the updating thread is defined as a *small task*, *game logic task*, and when the updating process is not parallelized at all (i.e. the not threaded figure). As previously demonstrated in Figure 5.11, this reiterates that the largest difference between *game logic task* and *small task* is that the main thread (i.e. the thread that renders particles) may have to wait on the *game logic task* if it is not finished by the end of the current frame. The *small task*, however, is not hindered by any restrictions; this task can be scheduled and ran concurrently alongside the rendering thread. A `ParticleUpdateTask` that is not scheduled as a background thread, however, forces the rendering thread to wait and thus no concurrent work is done.

types, and this section will delve into the different outcomes when ParticleUpdateTask is scheduled as a *small task*, a *game logic task*, and if it is not scheduled at all and runs on the main thread. The results of this study showed that using a *small task* to schedule a background thread to execute ParticleUpdateTask yielded much better results than two alternatives (i.e. scheduling a *game logic task* or not scheduling it at all).

A *small task* is considered to be a task with a short execution time such that it can be processed within a single frame, where "a single frame" is the time it takes for the engine to propagate all of its **Update** events from its *Application Systems*, *States*, *Entities*, *Components*, and so on (see Section 2.3.3). If a *small task* runs too long, then the task will have a negative impact on the overall performance of the program. A *game logic task* is a special kind of *small task* that is guaranteed to be executed prior to *main thread tasks*. It is easy to see, then, that long running *game logic tasks* will drastically affect the program's performance. *Main thread tasks* are also *small tasks* that are executed on the main thread after *game logic tasks*. *Main thread tasks* are executed by the main *Application* instance, and these are typically scheduled by background tasks to complete I/O jobs or any other job that needs to be sequential. All of these tasks are added to *Task Pools*, which are a collection of tasks that are performed by one of the worker threads owned by the pool. These pools will continue to perform tasks until they are deallocated and, if there are no more tasks to perform, they will sleep until another task is added.

Code Listing 19: Pseudocode showing how ChilliSource's *Task Scheduler* schedules *small tasks*, *game logic tasks*, and *main thread tasks*. This provides context on how the scheduler schedules tasks and, most importantly, how the *game logic task* uses the gameLogicCondition member variable to notify the main thread when all game logic tasks are executed (see Code Listing 20).

```
1  TaskScheduler::ScheduleTask(taskType, task)
2  {
3      switch(taskType)
4      {
5          // Add a task to the small thread task pool
6          case TaskType::small:
7              this.smallTaskPool.AddTask(task); break;
8
9          // Add a task to the main thread task pool
10         case TaskType::mainThread:
```

```
11                this.mainThreadTaskPool.AddTask(task); break;
12
13            // Schedule the task as a small task, but be sure to notify when
14            // all game logic tasks have been completed
15            case TaskType::gameLogic
16                ScheduleTask(TaskType::small,
17                {
18                    task();
19                    if(--this.gameLogicTaskCount is 0)
20                        this.gameLogicCondition.notify();
21                });
22
23                this.gameLogicTaskCount++;
24
25                break;
26
27            // File and large tasks can also be scheduled, but they are outside
28            // of this paper's scope
29        }
30    }
```

Code Listing 20: Pseudocode showing how *game logic tasks* force the main thread to wait until they are finished before allowing the main thread to continue. See Code Listing 19 to see how the `gameLogicCondition` variable is used to notify the main thread that the game logic tasks are completed.

```
1   TaskScheduler::ExecuteMainThreadTasks()
2   {
3       // Use a lock, mutex, and condition variable to wait
4       // until we are notified to continue.
5       using lock(this.gameLogicMutex) as lock
6       {
7           while(this.gameLogicTaskCount is not 0)
8               this.gameLogicCondition.wait(lock);
9       }
10
11      this.mainThreadTaskPool.PerformTasks();
12  }
```

5.3.2 Code

Scheduling the different types of tasks is fairly straightforward; the task type is simply passed into the `ScheduleTask` method. This was not included in previous code listings in order to simplify them, but the code listings below illustrate the different ways the `ParticleUpdateTask` function was scheduled for this study. It should be noted that, by default, the *`Particle Effect Component`* schedules `ParticleUpdateTask` as a

small task. Thus, during this study, the Small Task data series was analogous to the Original data series in previous sections.

5.3.2.1 Small and Game Logic Task Type

Code Listing 21: Pseudocode showing how ParticleUpdateTask can be scheduled as a *small task* and as a *game logic task*.

```
ParticleEffectComponent::OnUpdate()
{
  // Get a struct containing copies (and pointers) to pass to the task
  copiedAtts = CopyUpdateAttributes();

  // Schedule a SMALL TASK
  if (currTaskType is TaskType::small)
  {
      ApplicationTaskScheduler.ScheduleTask(TaskType::small,
    {
      ParticleUpdateTask(copiedAtts);
    });
  }
  // Schedule a GAME LOGIC TASK
  else if (currTaskType is TaskType::gameLogic)
  {
      ApplicationTaskScheduler.ScheduleTask(TaskType::gameLogic,
    {
      ParticleUpdateTask(copiedAtts);
    });
  }
}
```

5.3.2.2 No Task Scheduled (Not Threaded)

The sequential version of ParticleUpdateTask was performed in a similar fashion to the other tasks in that a struct was passed to it and the ConcurrentParticleData object was still used. Although these things were not strictly needed, they were included primarily to ”level the playing field” so that the results from all three versions were easier to compare.

Code Listing 22: Pseudocode showing how ParticleUpdateTask was not scheduled and was simply run on the main thread.

```
ParticleEffectComponent::OnUpdate()
{
  // Get a struct containing copies (and pointers) to pass to the task
  copiedAtts = CopyUpdateAttributes();

  // Do not schedule a task
```

```
7    ParticleUpdateTask(copiedAtts);
8  }
```

5.3.3 Results

The following two scenarios were used when generating metrics for the small task type, the game logic task type, and the series without a task scheduled:

1. Scenario used for particle metrics (Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6)
   - Min: 0
   - Max: 50,000
   - Step: 5,000
   - Each game was 5 seconds long
   - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
   - This scenario ran with 1 and 10 particle effects
2. Scenario used for timing metrics (Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.10)
   - Min: 50,000
   - Max: 50,000
   - Each game was 5 seconds long
   - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
   - This scenario ran with 1 and 10 particle effects

This totals to 24 series of games that were executed in order to generate these figures.

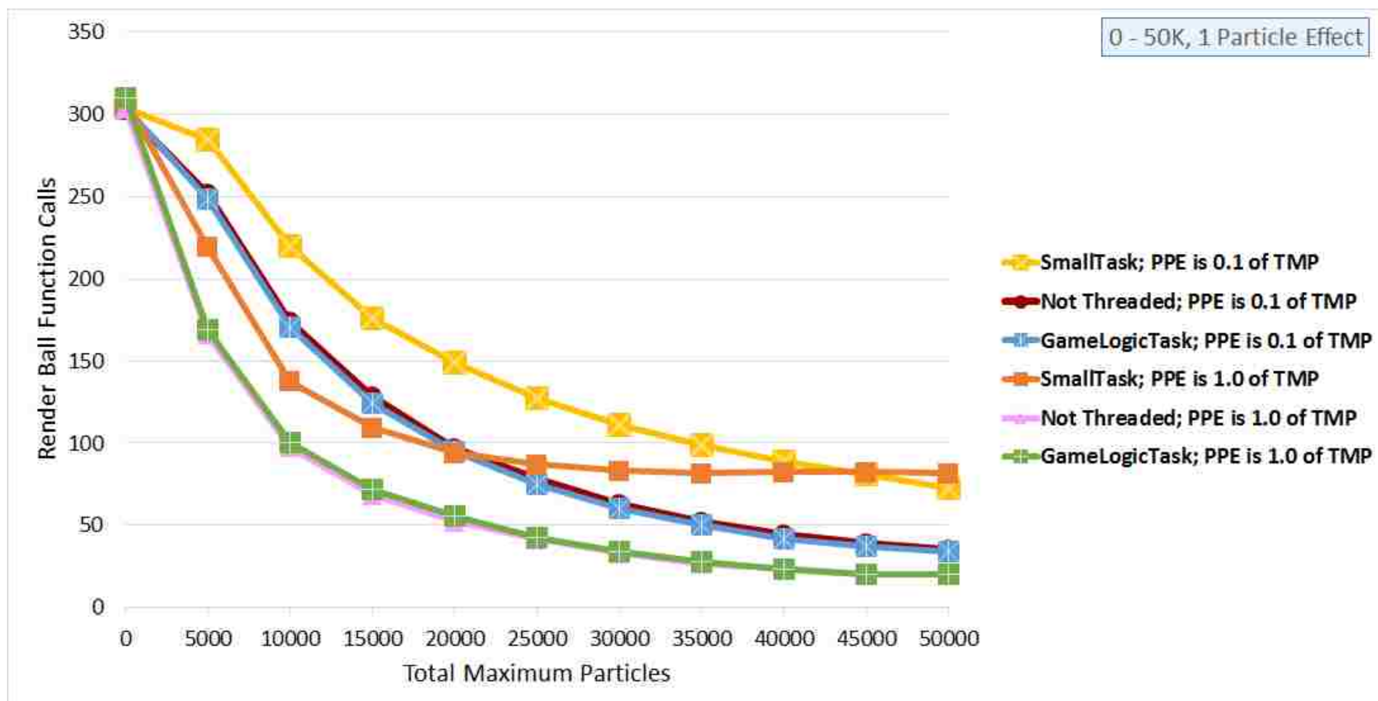

Figure 5.13 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows two things: the Not Threaded and Game Logic Task data series share nearly identical values, and the Small Task data series performs better (i.e. renders the ball more) than the other data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

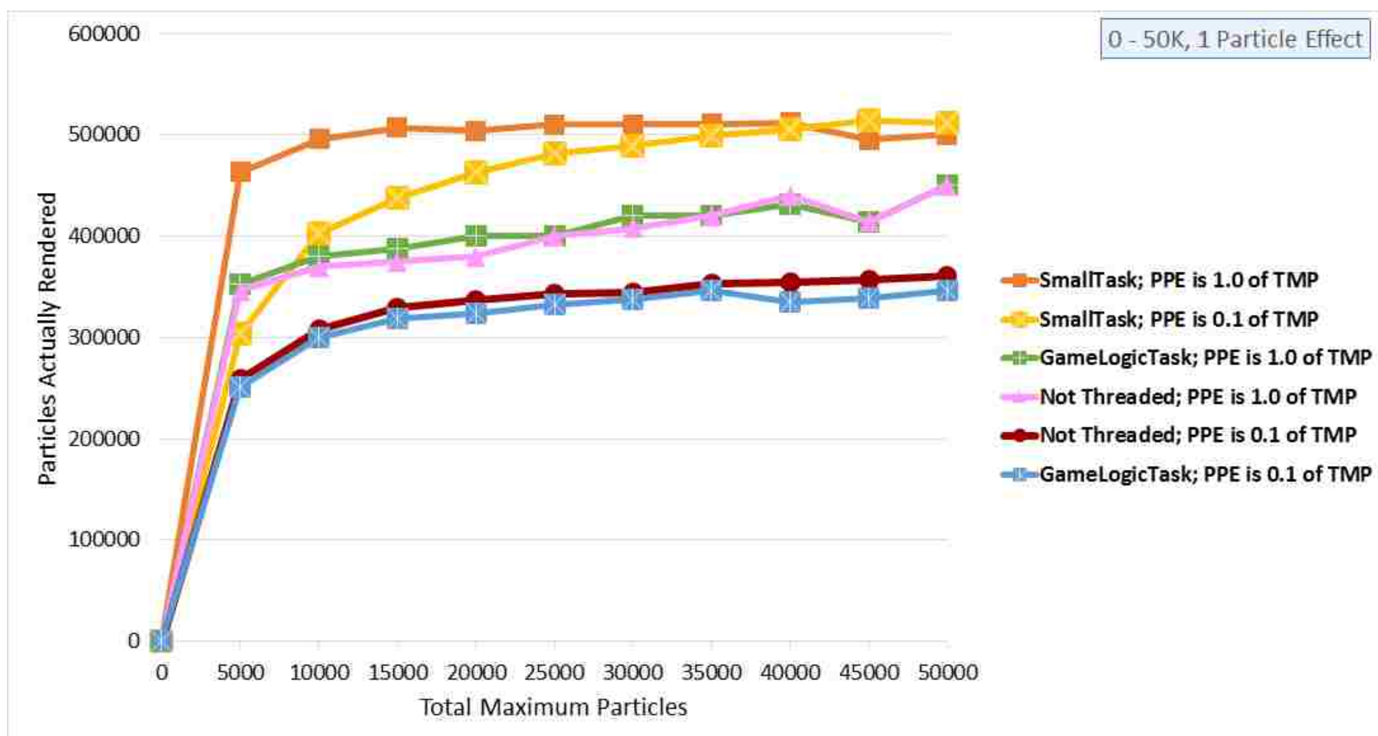

Figure 5.14 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. This also shows that the Original data series performed better than the Game Logic Task and Not Threaded data series by rendering more particles. Similar to Figure 5.13, the Game Logic and Not Threaded data series render nearly the same amount of particles.

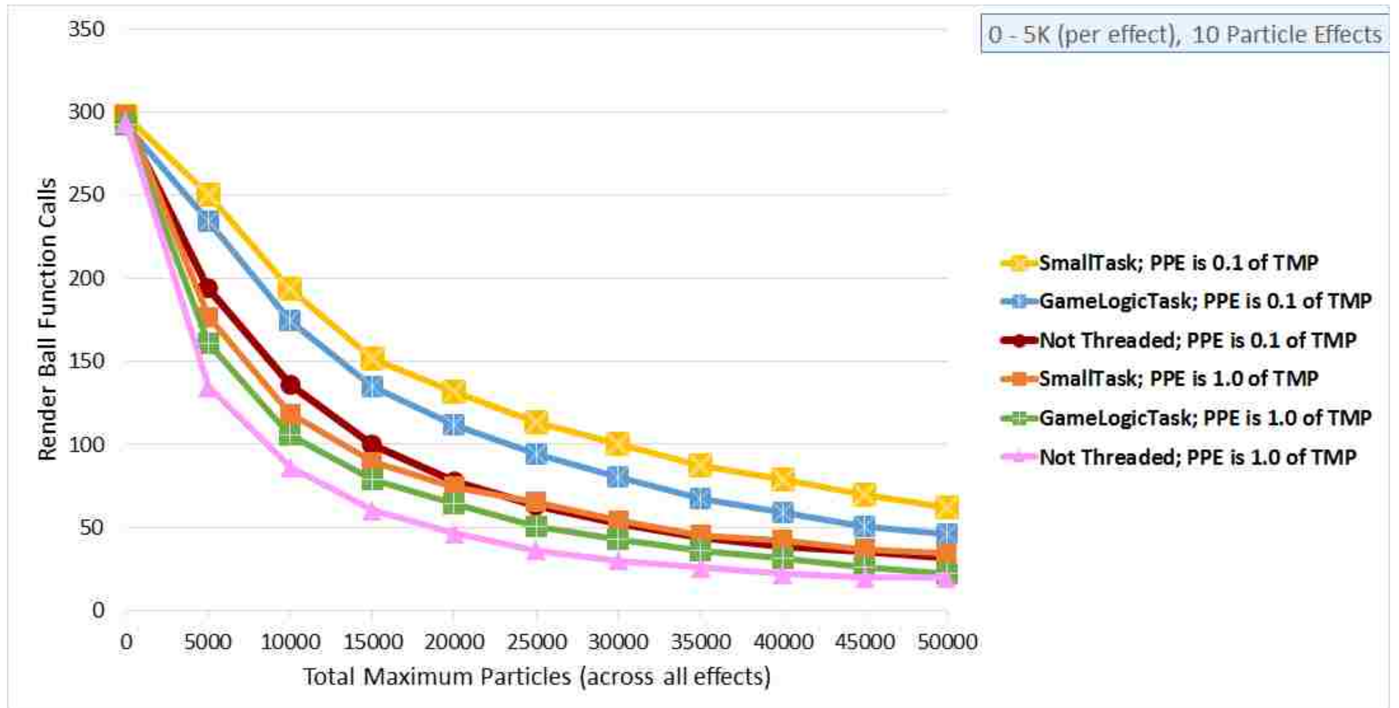

Figure 5.15 A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. Most notably this shows that not only the Original data series performed better than the Game Logic Task data series, but the Game Logic Task performed better than the Not Threaded data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

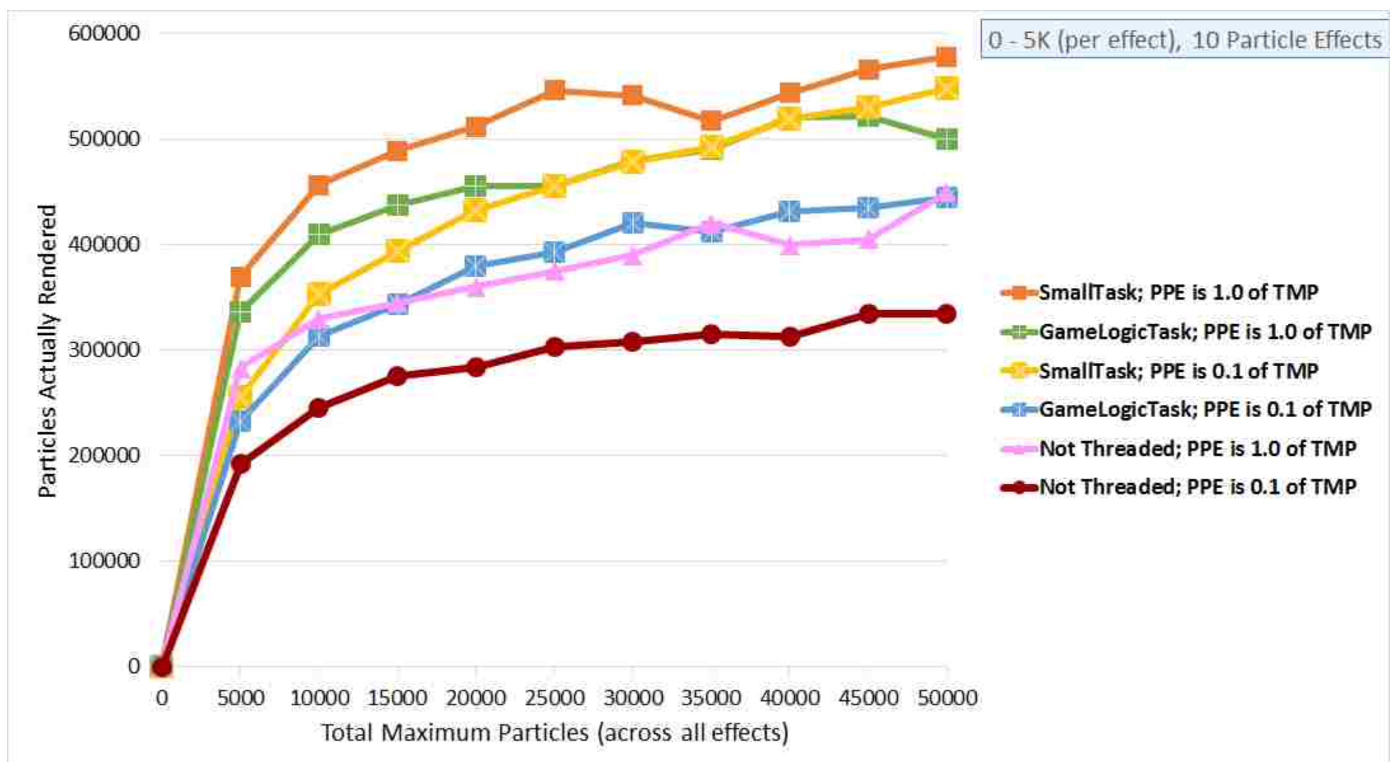

Figure 5.16 A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. This reiterates that the Game Logic Task data series out-performed the Not Threaded data series by rendering more particles, and that the Original data series out-performed both of the other data series.

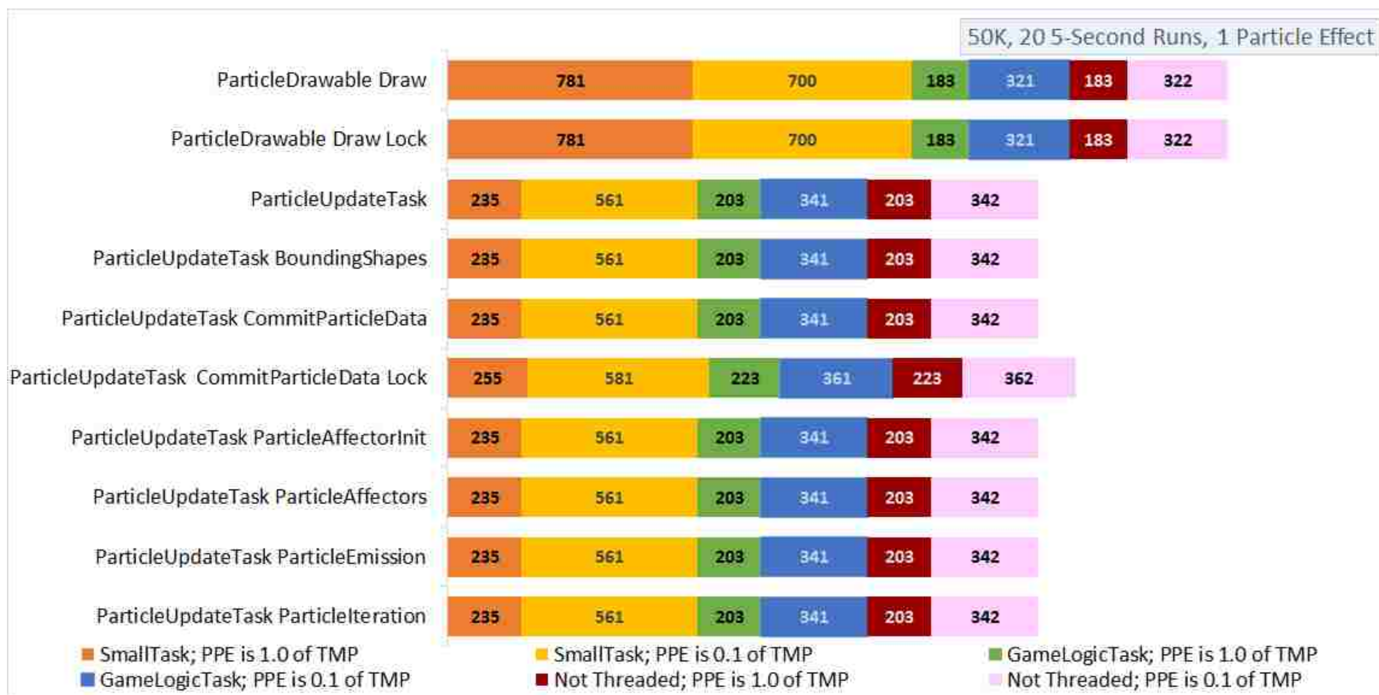

Figure 5.17 A graph showing the number of section calls for various sections of code for a single particle effect. This shows that the Game Logic and Not Threaded data series had about the same number of section calls and that the Original data series had more than both of the other data series for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games.

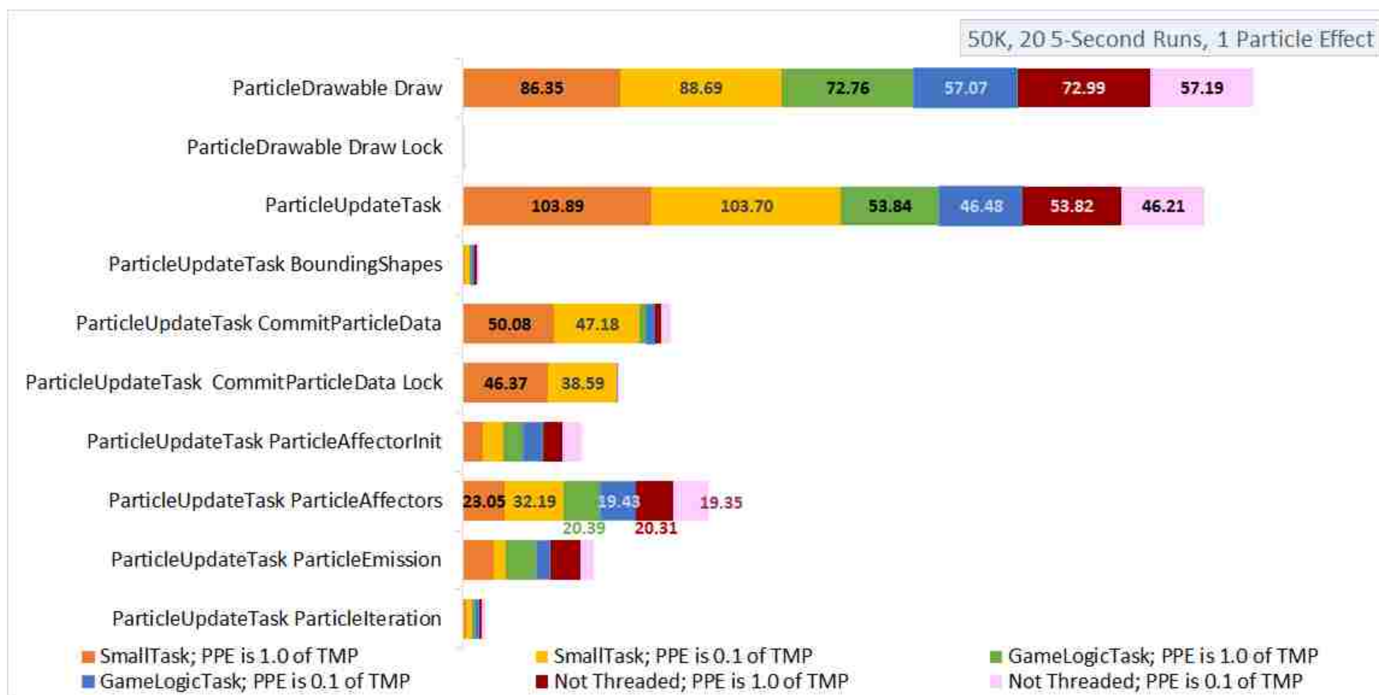

Figure 5.18 A graph showing the time spent in various sections of code for a single particle effect. This shows the Game Logic and Not Threaded data series shared the same run times for the multiple-burst (0.1 TMP) and single-burst (1.0 TMP) games. The Original Data series had longer run times, but this is because it had more opportunities to run (see the section call numbers in Figure 5.17) and not because of poor performance.

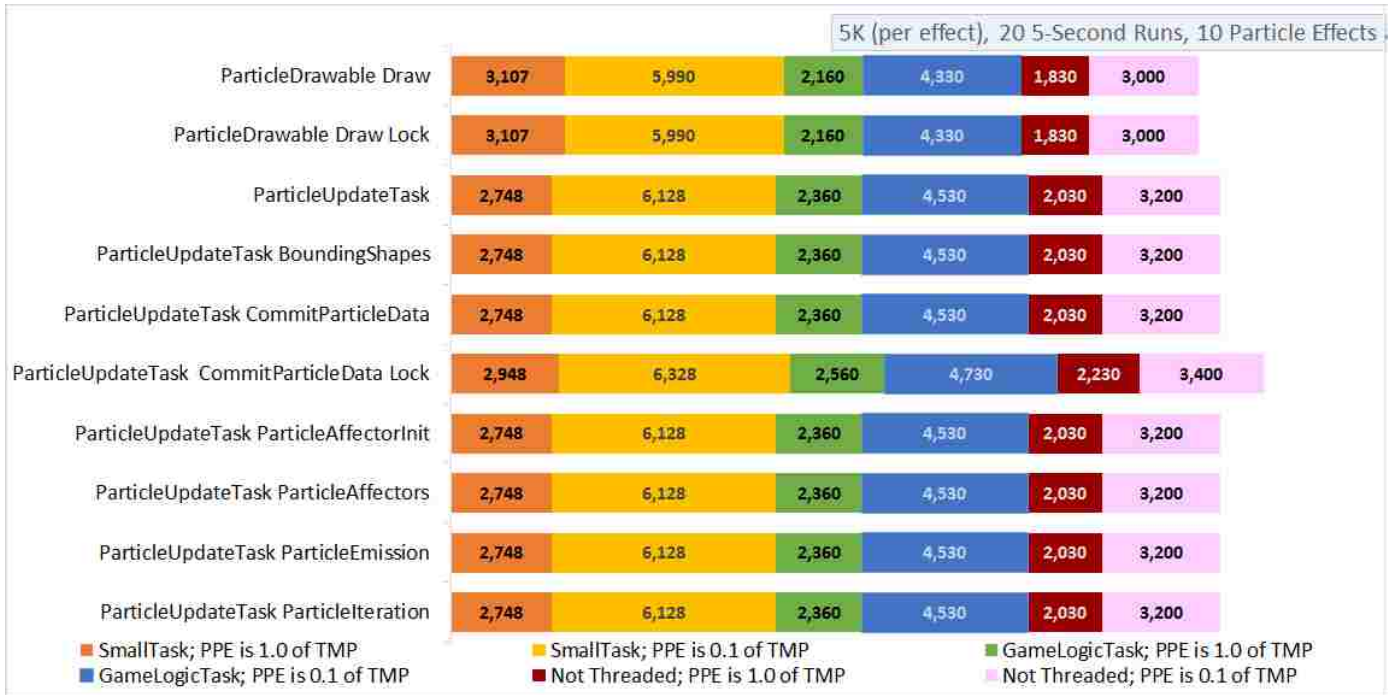

Figure 5.19 A graph showing the number of section calls for various sections of code for ten particle effects. Like in the particle metric graphs, this shows that the Game Logic Task data series performed better (i.e. more section calls) than the Not Threaded data series and that the Original data series had more section calls than them both.

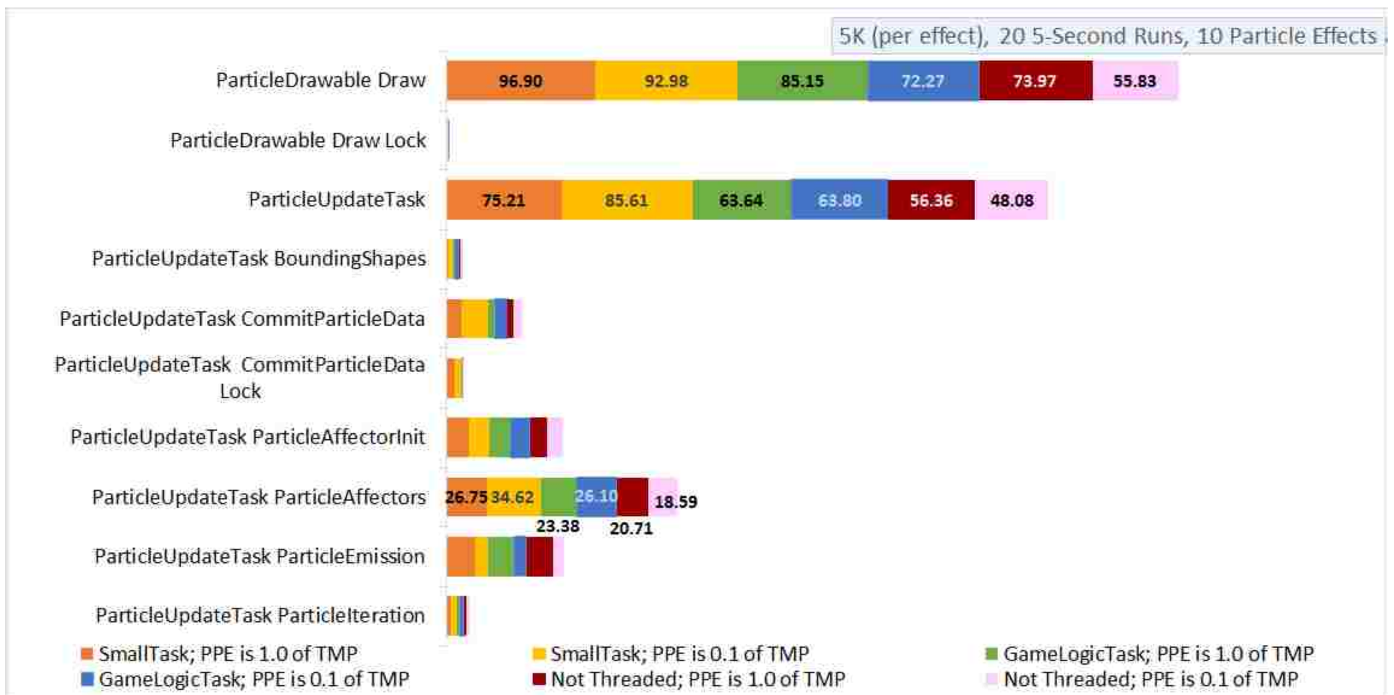

Figure 5.20 A graph showing the time spent in various sections of code for ten particle effects. This shows that the run times between the Game Logic and Not Threaded data series are not terribly different, but Not Threaded does have slightly lower run times than the Game Logic Task. Although a lower run time usually indicates better performance, in this case this is not true when we weigh this fact against Figures 5.15 and 5.16. This is because lower *render ball function calls* value results in choppy visuals, and that is not at all desirable to the end-user.

5.3.4 Discussion

For a single particle effect, the Not Threaded and GameLogicTask data series shared nearly identical values (Figures 5.13, 5.14, 5.17, and 5.18). They rendered the same amount of particles, they rendered the ball just as many times, and they spent the same amount of time in various parts of the code. This shows that, at least in the case of a single effect, the main thread waited about the same amount of time for the *game logic task* as if it was not even threaded.

For multiple particle effects, on the other hand, the overall performance of the Not Threaded data series was worse than the GameLogicTask data series (Figures 5.15, 5.16, 5.19, and 5.20). Although the timing metrics between the two data series were not too different, the particle metrics showed that the Not Threaded data series rendered fewer particles and the ball was not rendered as many times. These results revealed that being able to process multiple effects at the same time before executing the main thread tasks proved to be more efficient than processing them sequentially.

The results of this particular study demonstrated that scheduling background tasks may not necessarily be faster than its single-threaded implementation depending on how they are organized. Although it may seem obvious, it showed that being able to process many small tasks at once is more efficient than processing a single large task. As a software engineer, it is important to understand the lesson that these results teach. That is, forcing a multi-threaded solution on a problem that does not easily lend itself to doing many small tasks at once may not yield optimal results. In general, a multi-threaded program often requires a considerable amount of overhead and is challenging to debug. Thus, it is critical to structure multi-threaded programs such that it leverages these strengths. ChilliSource achieves this by creating a system that is built to update many `Particle Effect Components` at once in background threads.

5.4 Particles Failing to Render

5.4.1 Render Failure

`Particle Effect Components` sometimes failed to render particles during automated games. When this behavior was observed, I added particle metrics *particles actually rendered* and *particles actually emitted* to the instrumentation systems in order to numerically quantify (and, perhaps, understand) the failure. Two main variants of this failure emerged:

1. Particles were slow to emit, and it may look like it never emitted if the game's duration was too short
2. Particles never emitted even if the game's duration was lengthened significantly

The first variant usually exhibited a low *render ball function call* metric, i.e. a low frames per second, while the second variant did not seem to suffer from a performance hit in the slightest. The first kind was, in a way, expected for particle effects with a large number of particles simply because the renderer was slowed by the sheer number of iterations it was forced to make every update cycle. The second kind, on the other hand, was puzzling because it was not expected behavior. For reasons that will be explained later on in this chapter, the particle effect never emitted any particles in the second variant because the effect met two conditions:

- The effect's duration elapsed
- There were no active particles

This was found to be due to the particle effect having a small duration, a large number of particles, and it was not "looping". Since this was an important distinction between the results of the two variants during this study, it is important to understand what "looping" means in the context of a `Particle Effect Componant`. A looping particle effect will continue to draw and update itself even after its duration has elapsed, while a non-looping particle effect will stop drawing and updating once its duration has elapsed. Figure 5.21 illustrates the difference between looping and non-looping particle effects.

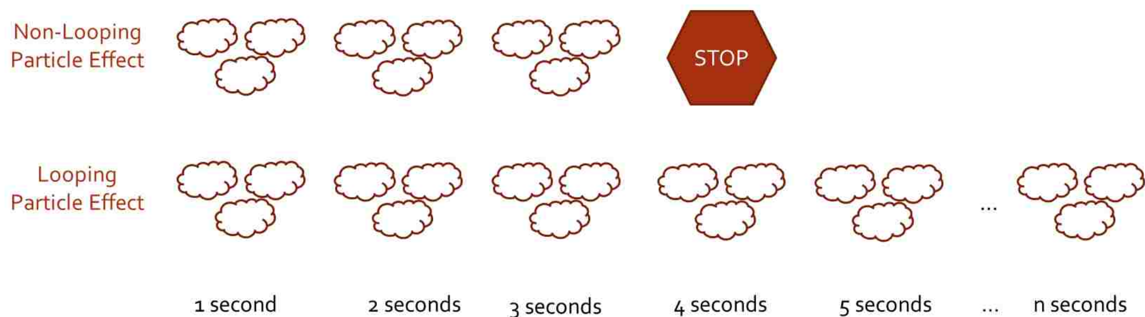

Figure 5.21: Demonstrates the difference between looping and non-looping particle effects that both have a duration of 3 seconds. Each particle effect emits once every second, but the non-looping particle effect will stop emitting after 3 seconds have elapsed. It's important to note that, although the non-looping particle effect has stopped emitting, ChilliSource will continue to render any active particles on the screen from previous emissions. The engine will only clean up the particle effect if there are no longer any active particles on the screen.

5.4.2 Emission Failure Source

In order to demonstrate and understand this particle failure, metrics were generated using the same two scenarios as used in the past:

1. Scenario used for particle metrics (Figure 5.22, Figure 5.23, Figure 5.24, Figure 5.25)
    - Min: 0
    - Max: 50,000
    - Step: 5,000
    - Each game was 5 seconds long
    - This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
    - This scenario ran with 1 and 10 particle effects
2. Scenario used for timing metrics (Figure 5.26, Figure 5.27, Figure 5.28, Figure 5.29)
    - Min: 50,000
    - Max: 50,000
    - Each game was 5 seconds long

- This scenario ran when *particles per emission* (PPE) was 10% and 100% of *total maximum particles* (TMP) or, in other words, when the particle effect had multiple-bursts and a single burst.
- This scenario ran with 1 and 10 particle effects

These two scenarios were both run for non-looping and looping particle effects[3]. This totals to 16 series of games that were executed in order to generate these figures.

From these results (Figures 5.22, 5.23, 5.26, 5.27), it can be clearly seen that the "Non-Looping; PPE is 1.0 of TMP" data series for a single particle effect demonstrates the catastrophic emission failure. At about 30,000 total maximum particles, zero particles were rendered and, as a result, the *render ball function calls* skyrocketed since the renderer was not focusing on rendering particles and instead had more opportunities to render the ball since the drawing step took less time (see Figure 5.27). What was also interesting about this series was the extremely short amount of time that was spent in `ParticleUpdateTask`. The other data series took up approximately 100 seconds when running `ParticleUpdateTask`. Instead, it only spent about 27 seconds in `ParticleUpdateTask` (see Figure 5.27).

The other data series from Figures 5.22, 5.23, 5.26, 5.27 did not experience the same catastrophic failure as "Non-Looping; PPE is 1.0 of TMP". The only one that came close was the "Non-Looping; PPE is 1.0 of TMP" data series for multiple particle effects (see Figures 5.24, 5.25, 5.28, 5.29). For this data series, the *particles actually rendered* dipped and the *render ball function calls* rose at 40,000 total maximum particles. Thus, I focused on that **non-looping** particle effects with a large number of particles per emission.

5.4.3 Emission Failure Solution

Discovering why this failure occurred was a multi-step process. The automated game was set to run with a single effect with PPE at 1.0 TMP and a particle value of at least 30,000.

[3]The two data series, Non-Looping and Looping, were *`Particle Effect Components`* that have `isLooping` set to false and true, respectively.

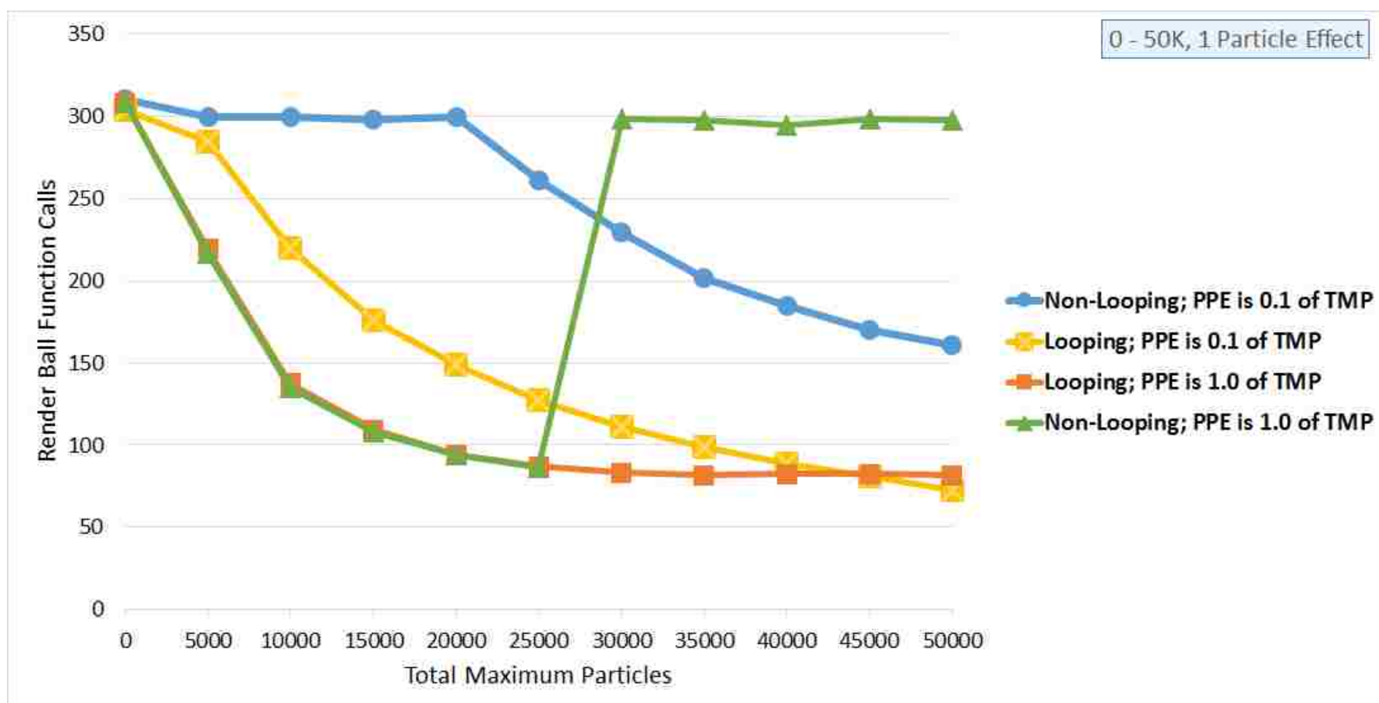

Figure 5.22 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows that, overall, the Non-Looping data series performs better (or renders more particles) than the Looping (i.e. Original) data series. However, it is important to pair this fact with two other observations: (1) Figure 5.23 shows that the Non-Looping data series renders considerably less particles, and (2) at about 30,000 TMP there is spike in *render ball function calls*.

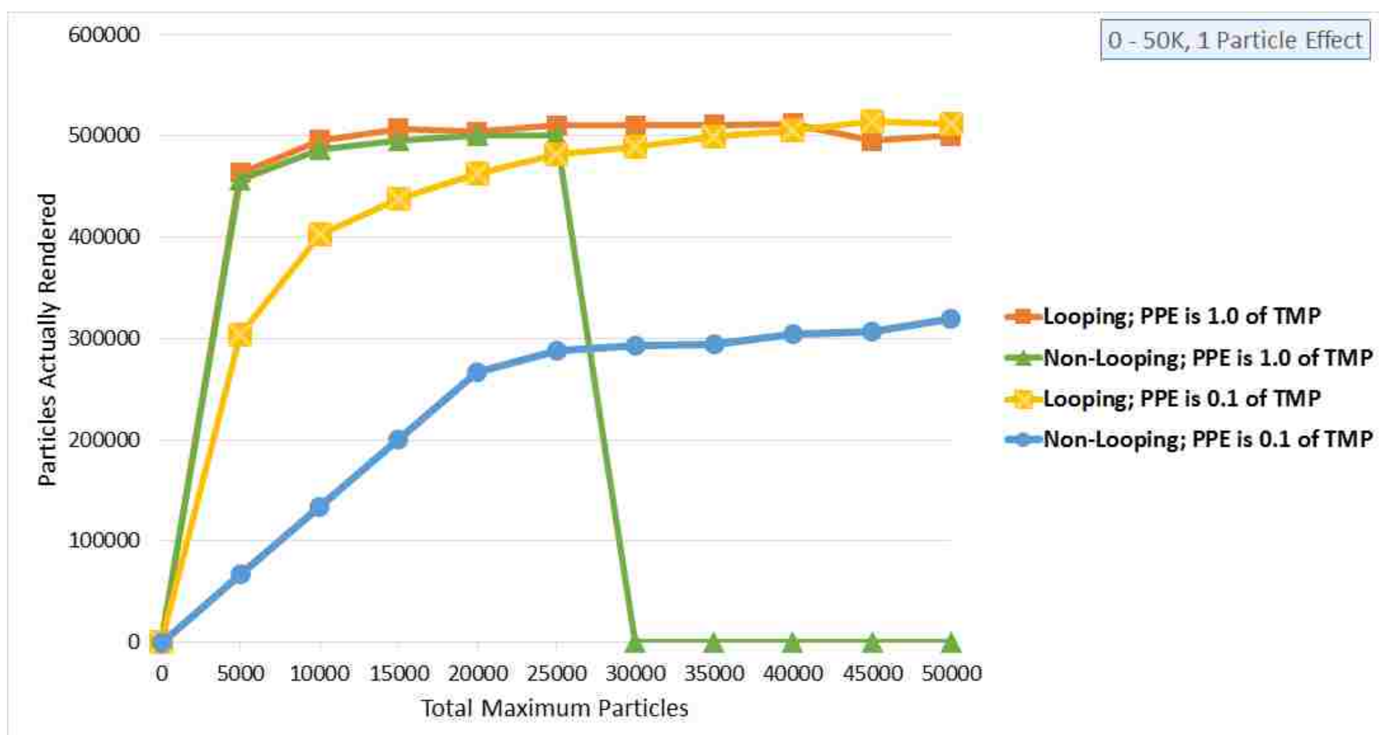

Figure 5.23 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. As mentioned in Figure 5.22, this graph illustrates that the Non-Looping data series, overall, does not render as many particles as the Looping data series. At 30,000 TMP, the Non-Looping data series does not render any particles at all.

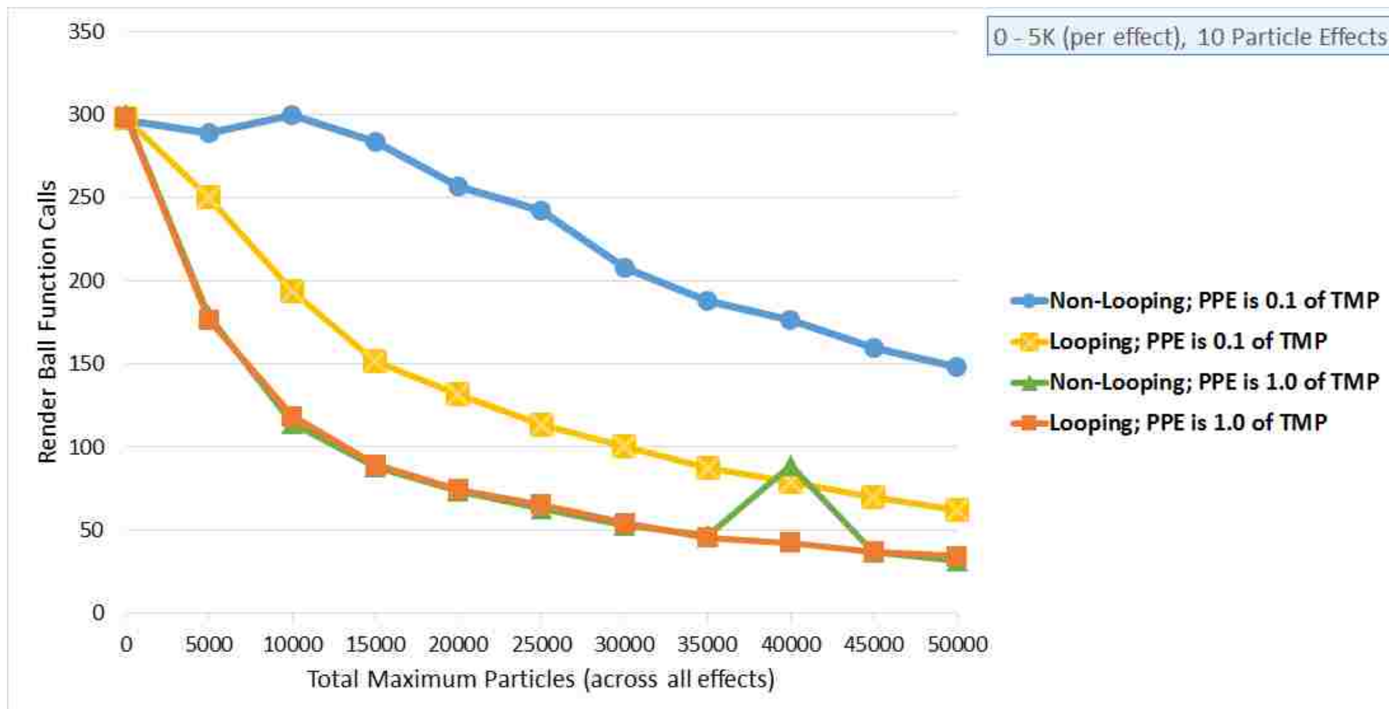

Figure 5.24 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows two things: (1) The single-burst (or 1.0 TMP) games for Looping and Non-Looping data series renders the ball nearly the same number of times (except at 40,000 TMP the performance spikes when the number of rendered particles drops- see Figure 5.25), and (2) The multiple-burst (or 0.1 TMP) games for the Non-Looping data series renders the ball significantly more times.

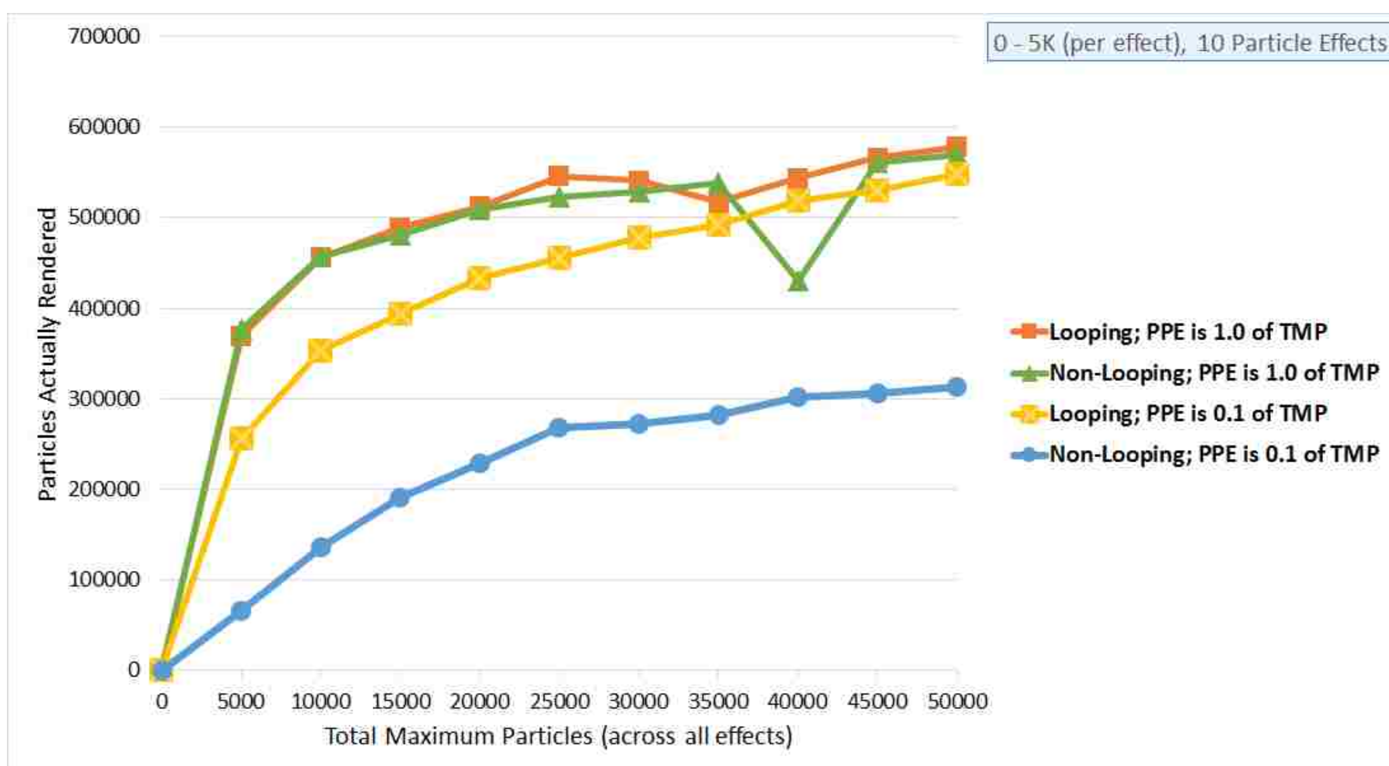

Figure 5.25 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. As mentioned in Figure 5.24, this shows that the Non-Looping data series renders nearly as many particles as the Looping data series for the single-burst (or 1.0 TMP) games (except at 40,000 TMP when the number of rendered particles drops). It also shows that the Non-Looping data series does not render as many particles as the Looping data series for the multiple-burst (or 0.1 TMP) games.

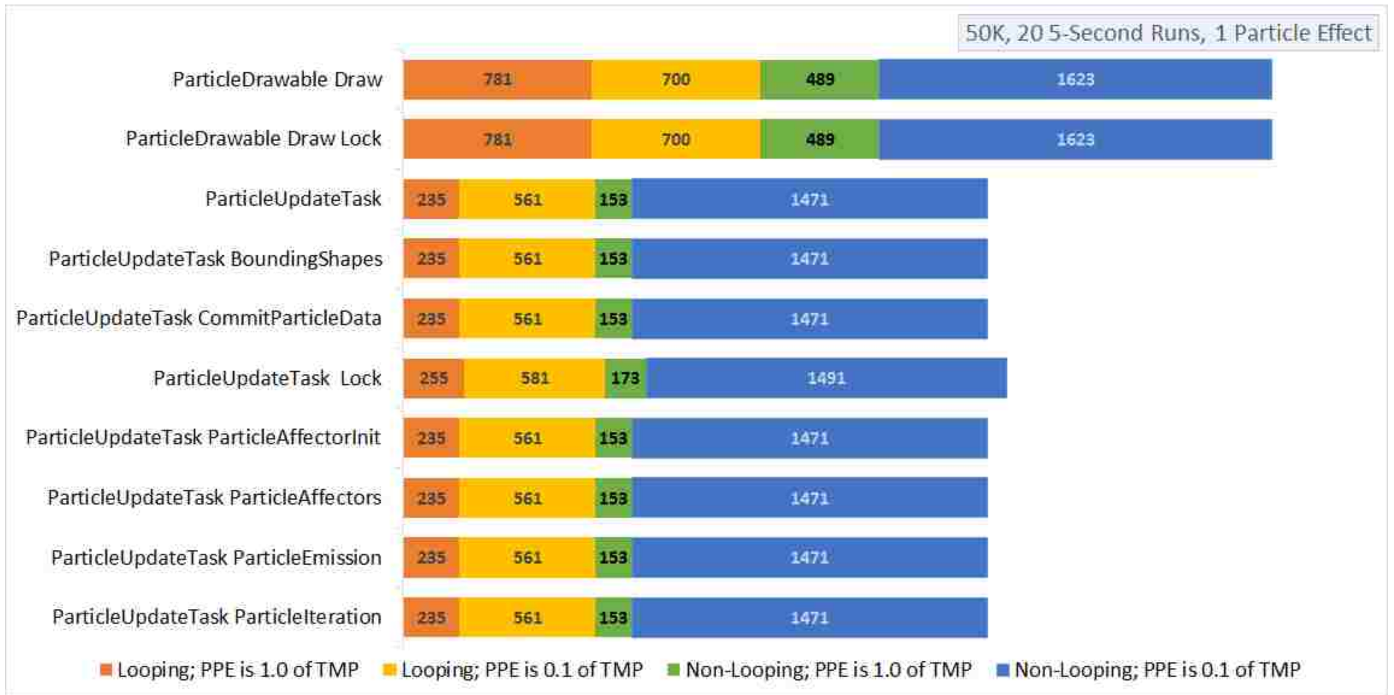

Figure 5.26 A graph showing the number of section calls for various sections of code for a single particle effect. This shows that: (1) The Non-Looping data series in multiple-burst (or 0.1 TMP) games have a significantly higher section call count than the Looping data series, and (2) The Non-Looping data series in single-burst (or 1.0 TMP) games have a lower section call count than the Looping data series.

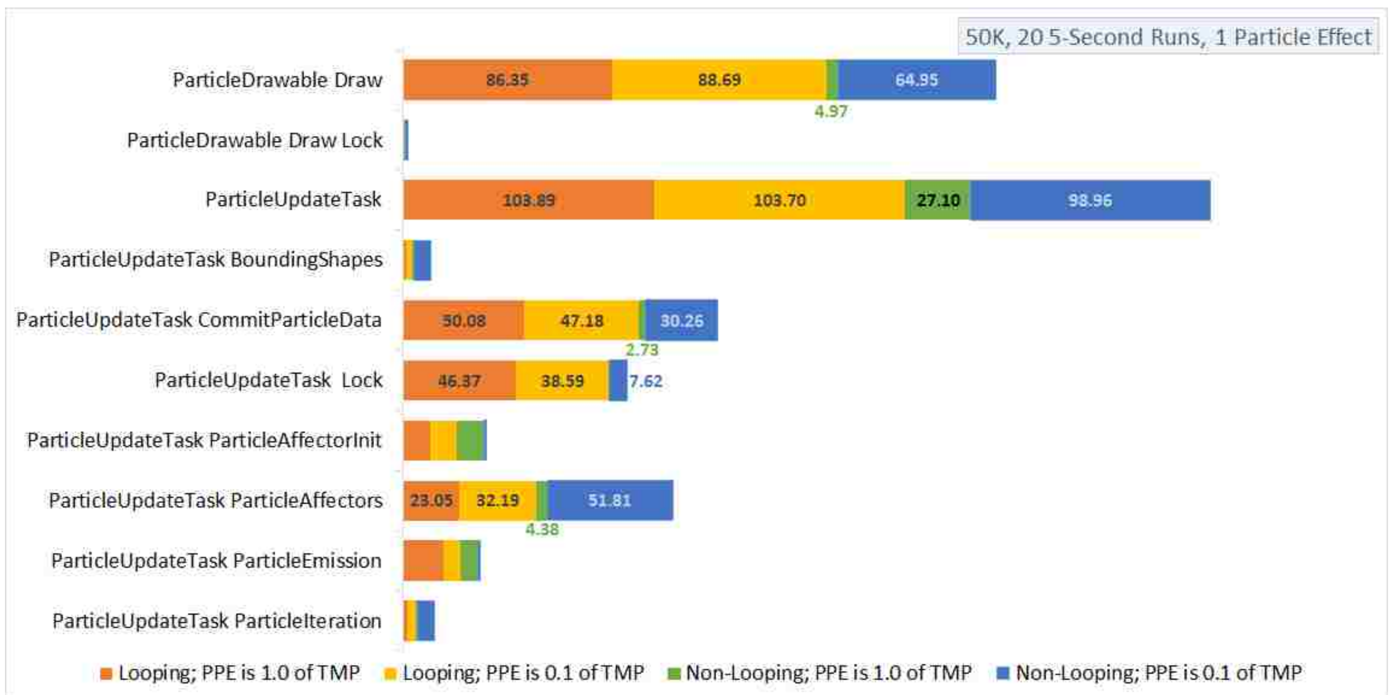

Figure 5.27 A graph showing the time spent in various sections of code for a single particle effect. Similar to Figure 5.26, this shows that: (1) The Non-Looping data series in multiple-burst (or 0.1 TMP) games have a lower run time than the Looping data series, and (2) The Non-Looping data series in single-burst (or 1.0 TMP) games have a significantly lower run time than all of the data series and games.

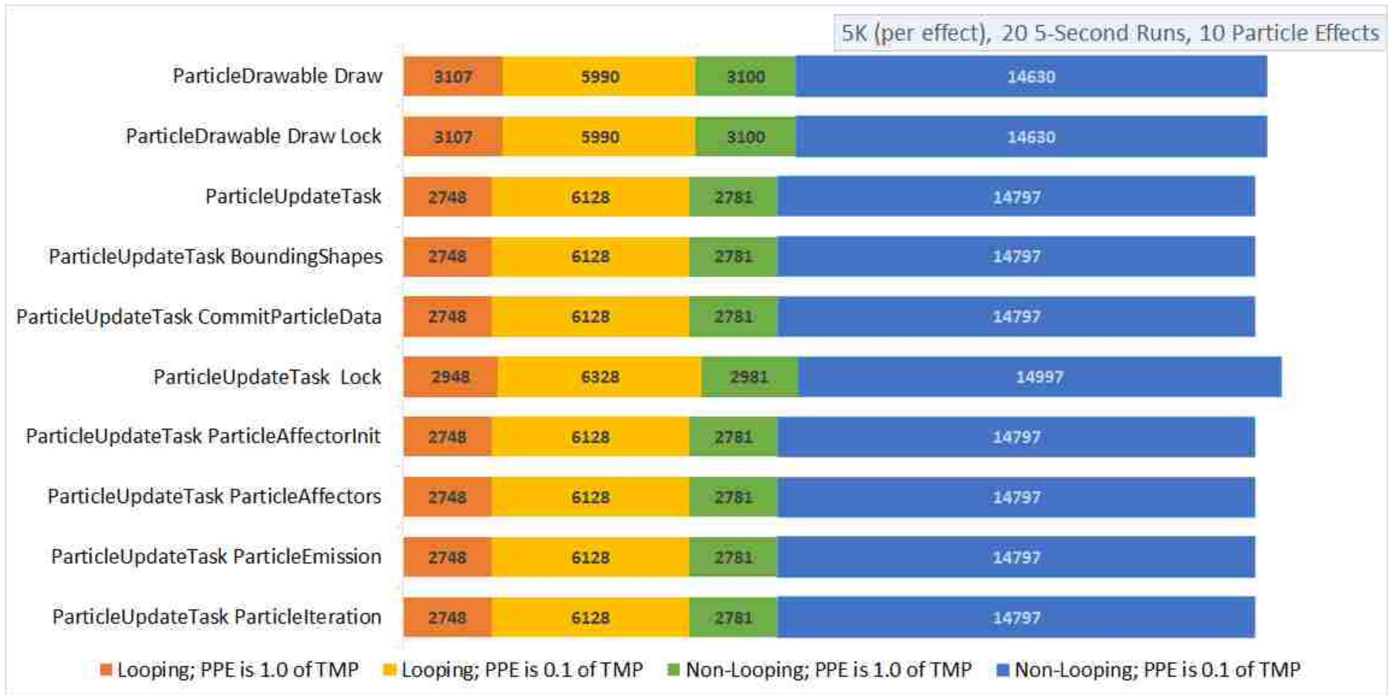

Figure 5.28 A graph showing the number of section calls for various sections of code for ten particle effects. This shows that: (1) The Non-Looping data series in multiple-burst (or 0.1 TMP) games have a significantly higher section call count than the Looping data series, and (2) The Non-Looping data series in single-burst (or 1.0 TMP) games approximately have the same section call counts as the Looping data series.

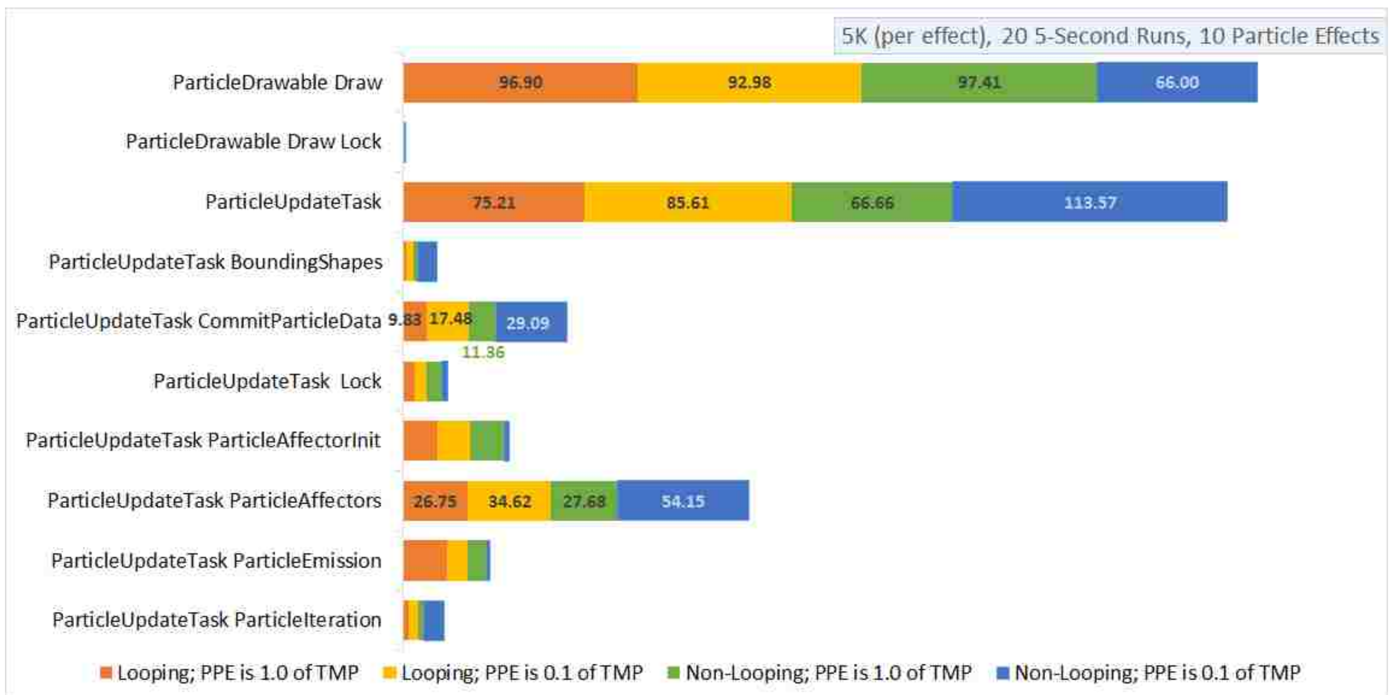

Figure 5.29 A graph showing the time spent in various sections of code for ten particle effects. This shows that: (1) The Non-Looping data series in multiple-burst (or 0.1 TMP) games have a higher run time for particle update functions and a lower run time for particle draw functions, and (2) The Non-Looping data series in single-burst (or 1.0 TMP) games have similar run times as the Looping data series.

Since this seemed to be a rendering issue, breakpoints were set in the `Render` function within the *`Particle Effect Component`* class. I discovered that the `Render` function inside of a *`Particle Effect Component`* was never called during a game with a single effect with PPE at 1.0 TMP (single-burst) and a particle value of at least 30,000. To further investigate, breakpoints were placed within the renderer itself when it iterated over *`Render Components`* and, specifically, when it iterated over a *`Particle Effect Component`*. It turned out that the effect was being "culled", or ignored, because the bounding information associated with the effect was set to its initial value: `INTEGER_MAX`. Since the actual, non-default value of its bounding information was set inside of `ParticleUpdateTask`, this suggested that `ParticleUpdateTask` was not being called anymore. Placing breakpoints within the `Update` method in the *`Particle Effect Component`* revealed that it called the `Stop` function and ceased to schedule `ParticleUpdateTask`. As mentioned earlier, this was called because the effect met two conditions:

- The effect's duration elapsed
- There were no active particles

These conditions were met because `ParticleUpdateTask` had not updated or activated any particles within the effect's duration time, which was 1 second. This can happen if the main thread that schedules the `ParticleUpdateTask` runs again before the `ParticleUpdateTask` finishes its first update. This results in the main thread stopping the particle effect because it thinks that the particle effect is done running (i.e. it saw that there were no active particles and the effect's duration elapsed). Thus, it can be inferred that a non-looping particle effect coupled with a small duration and a large number of particles resulted in particle failure. Increasing the duration of a such a particle effect should fix the problem.

The following figures (Figures 5.30, 5.31, 5.32, 5.33, 5.34, 5.35, 5.36, and 5.37) present metrics that compare the emission failure from above with the solution to that failure. The metrics from this "Solution" data series are the result of running the same scenarios as above

(see Section 5.4.2) with a non-looping particle effect, but this non-looping particle effect had an increased duration time of 120 seconds[4]. An examination of these results shows that this solution did indeed solve the problem. Particles were rendered from 0 to 50,000 total maximum particles, and a healthy amount of time was spent within `ParticleUpdateTask`, as expected.

In essence, this study has revealed a bug that ChilliSource should fix. Since the failure occurred because the `ParticleUpdateTask` did not fully execute once by the time the particle effect's duration elapsed, the failure would not happen if the engine allowed the effect to update at least once before cleaning up and stopping the effect.

[4]Since a duration of any given game was only 5 seconds, a particle effect duration of 120 seconds is overkill. I set the effect's duration to this high number to be certain that it was a small duration number that was causing the particle emission failure behavior. Note that this is separate from the lifetime of a particle itself; the duration that is being referred to here is the duration of the particle effect as a whole and not the lifetime of a particle individually.

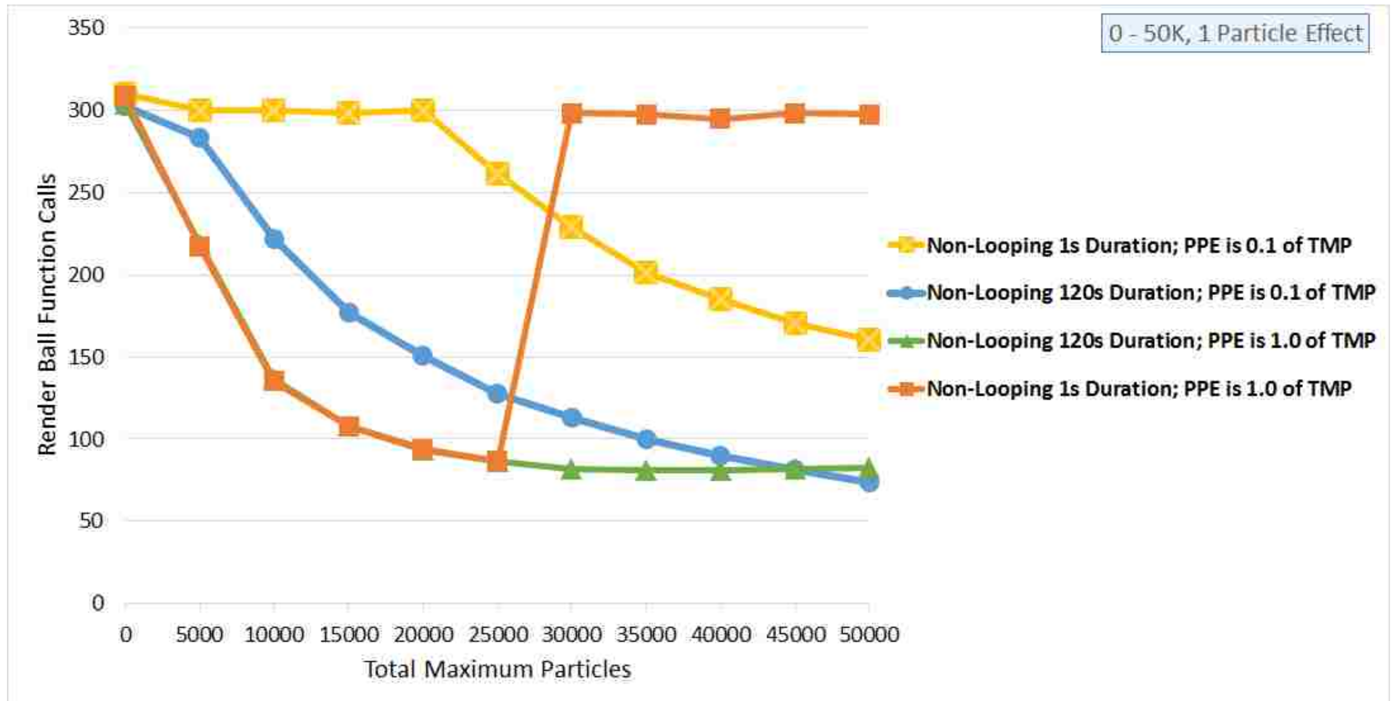

Figure 5.30 A graph showing *render ball function calls* over *total maximum particles* for a single particle effect. This shows that the Non-Looping 120s Duration data series (the one with an increased duration of 120 seconds) does not spike in performance at 30,000 TMP. This is a good thing because it means that the renderer was spending some of its time rendering particles (see Figure 5.33).

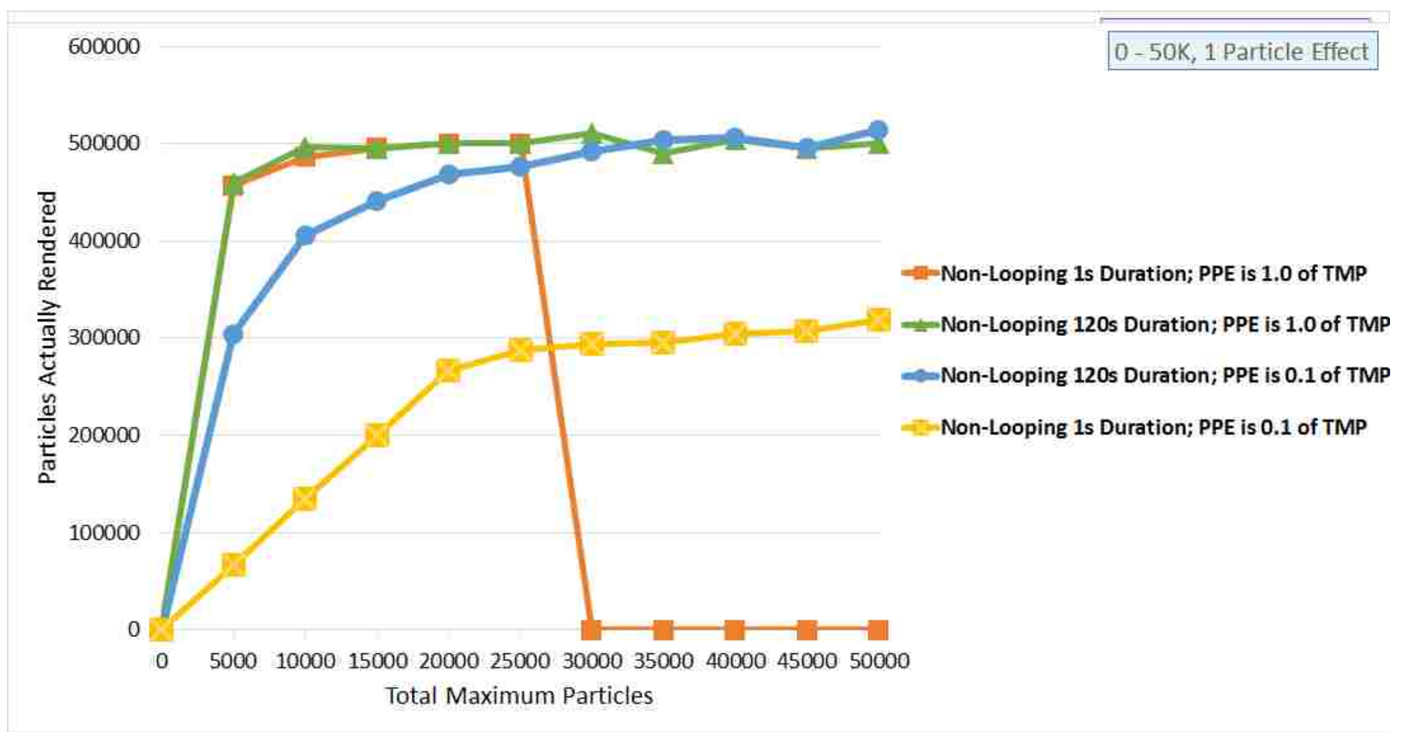

Figure 5.31 A graph showing *particles actually rendered* over *total maximum particles* for a single particle effect. This shows that the Non-Looping 120s Duration data series (the one with an increased duration of 120 seconds) does not stop rendering particles to the screen at 30,000 TMP, which is the desired outcome.

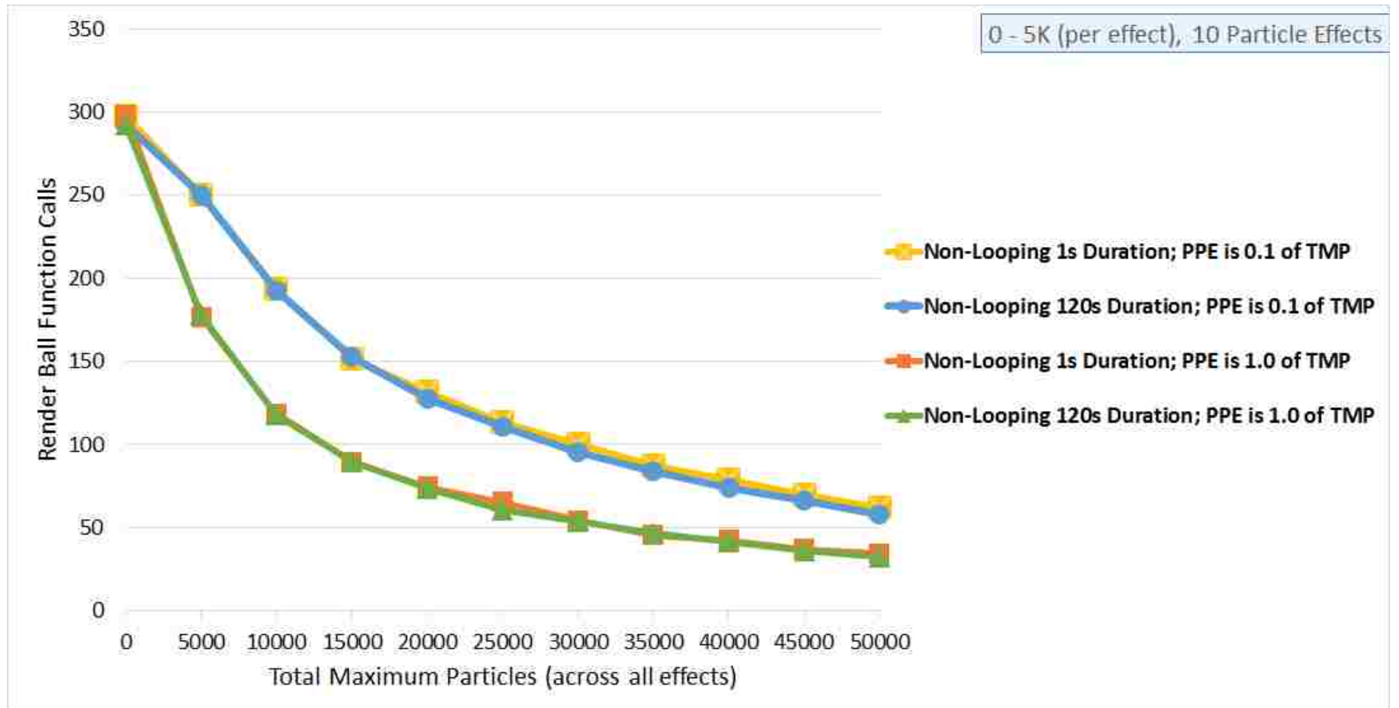

Figure 5.32 A graph showing *render ball function calls* over *total maximum particles* for ten particle effects. Since this failure did not occur for many particle effects, we see here that the Solution and Non-Looping 1s Duration data series render the ball about the same number of times.

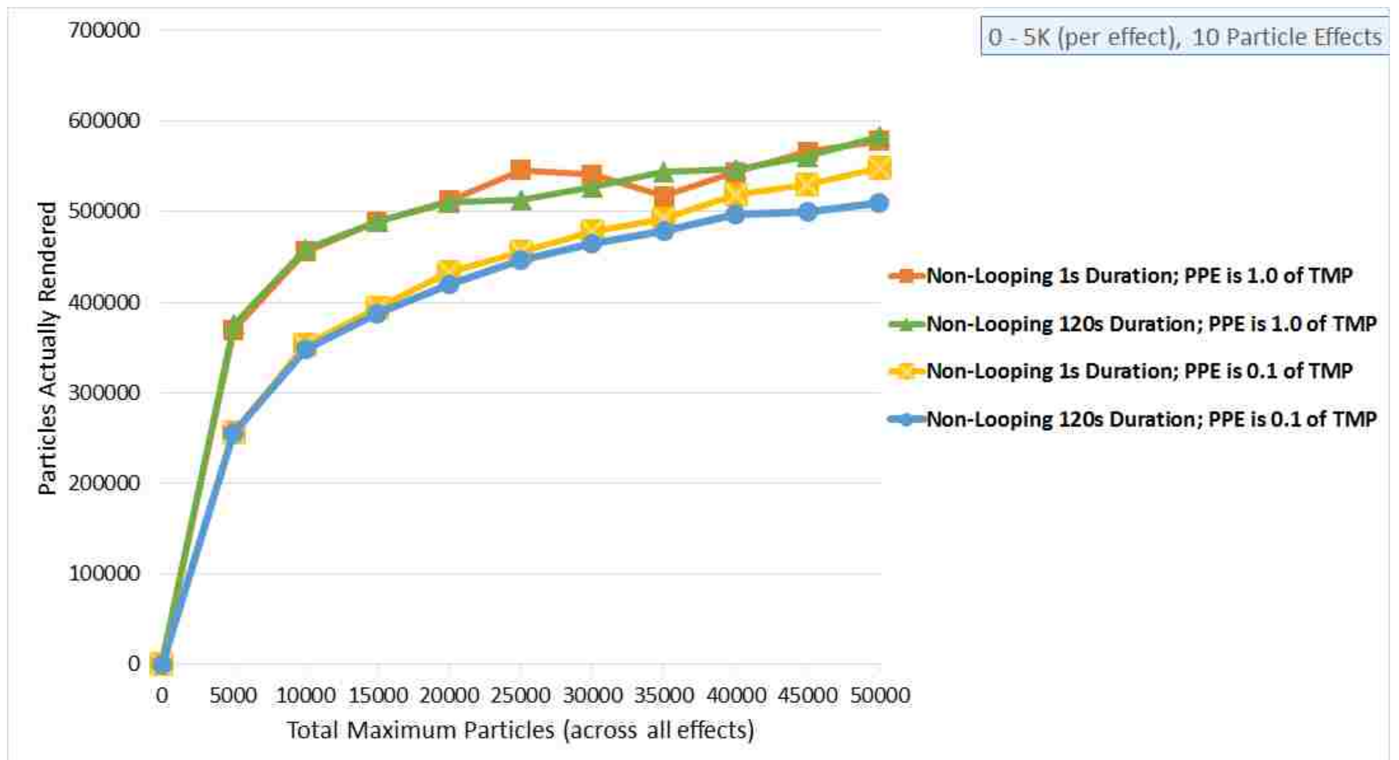

Figure 5.33 A graph showing *particles actually rendered* over *total maximum particles* for ten particle effects. Similar to Figure 5.32, this shows that the Solution and Non-Looping 1s Duration data series render about the same number of particles.

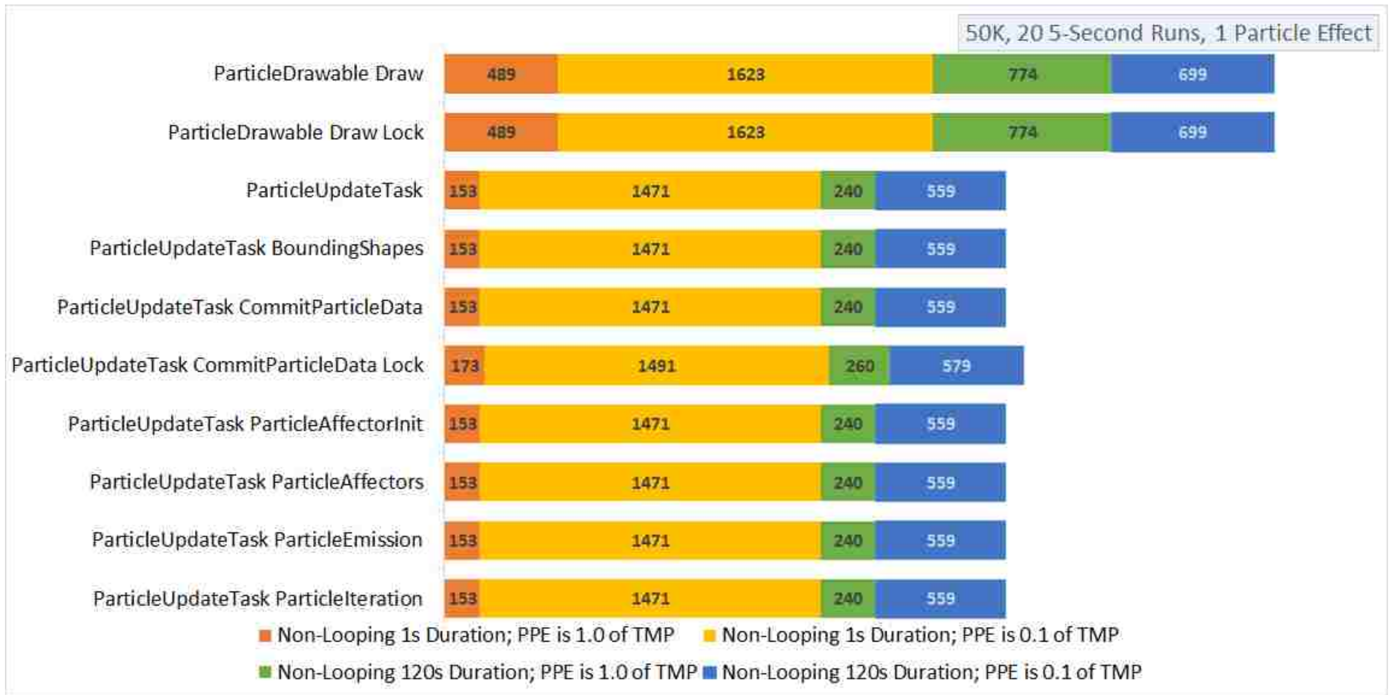

Figure 5.34 A graph showing the number of section calls for various sections of code for a single particle effect. This shows that the Non-Looping 120s Duration data series has fewer section calls than the Non-Looping 1s Duration data series.

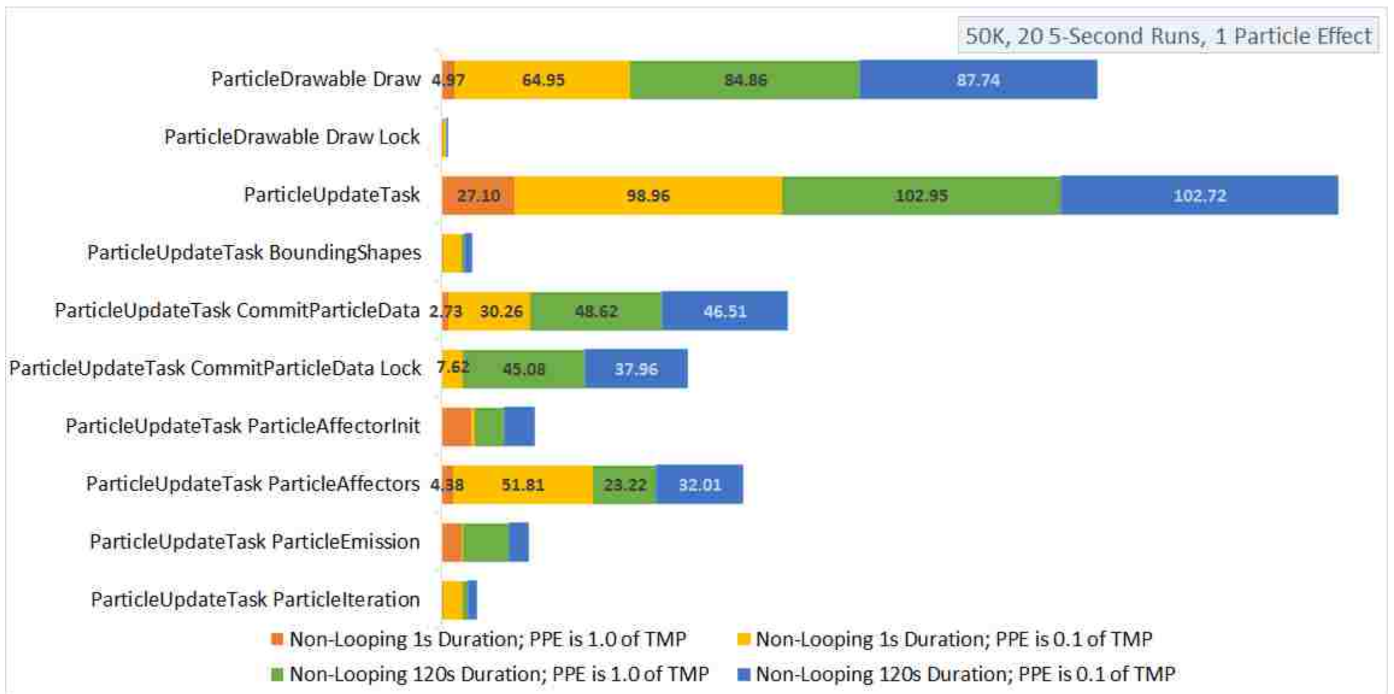

Figure 5.35 A graph showing the time spent in various sections of code for a single particle effect. This shows that the Non-Looping 120s Duration data series has overall longer run times than the Non-Looping 1s Duration data series; this is because the `Particle Effect Component` is actually being rendered and processed instead of being stopped early on in the game.

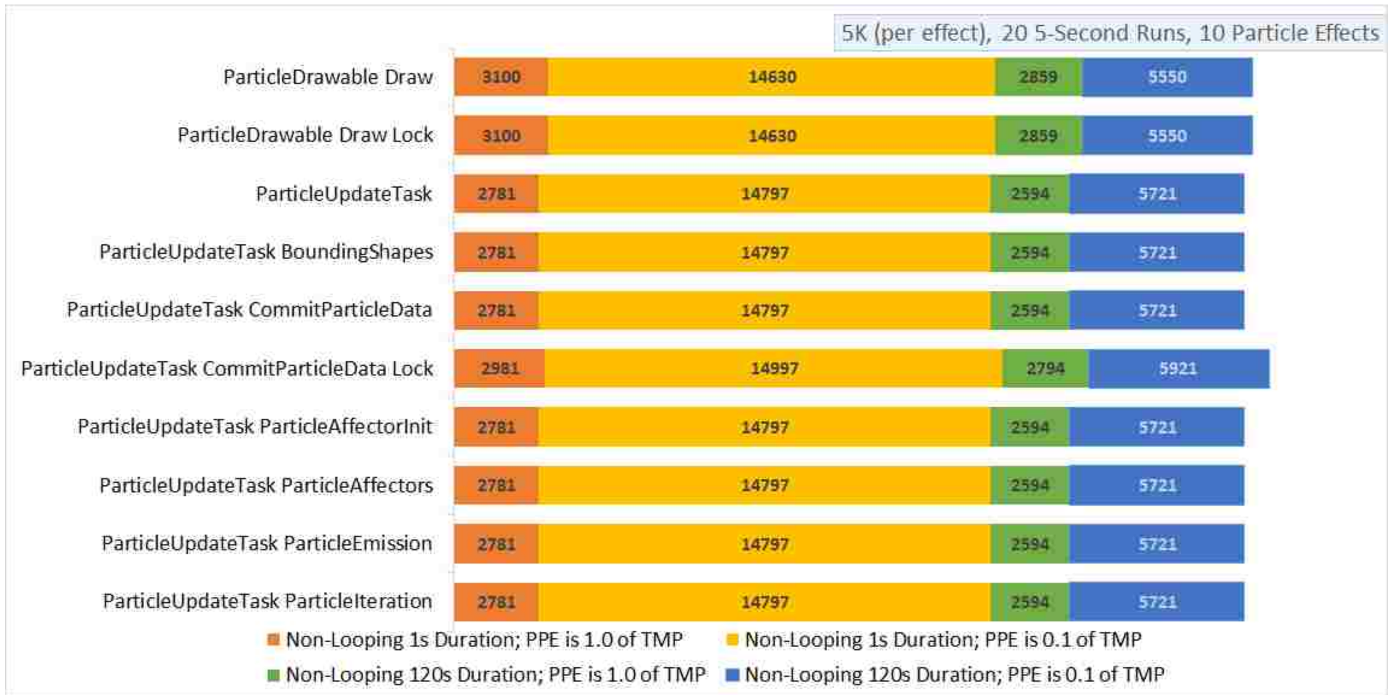

Figure 5.36 A graph showing the number of section calls for various sections of code for ten particle effects. This shows: (1) The multiple-burst (or 0.1 TMP) games for the Non-Looping 1s Duration data series has significantly more section calls than the Non-Looping 120s Duration data series, and (2) The single-burst (or 1.0 TMP) games for the Non-Looping 1s Duration data series has a little more section calls than the Non-Looping 120s Duration data series, but their values are fairly similar.

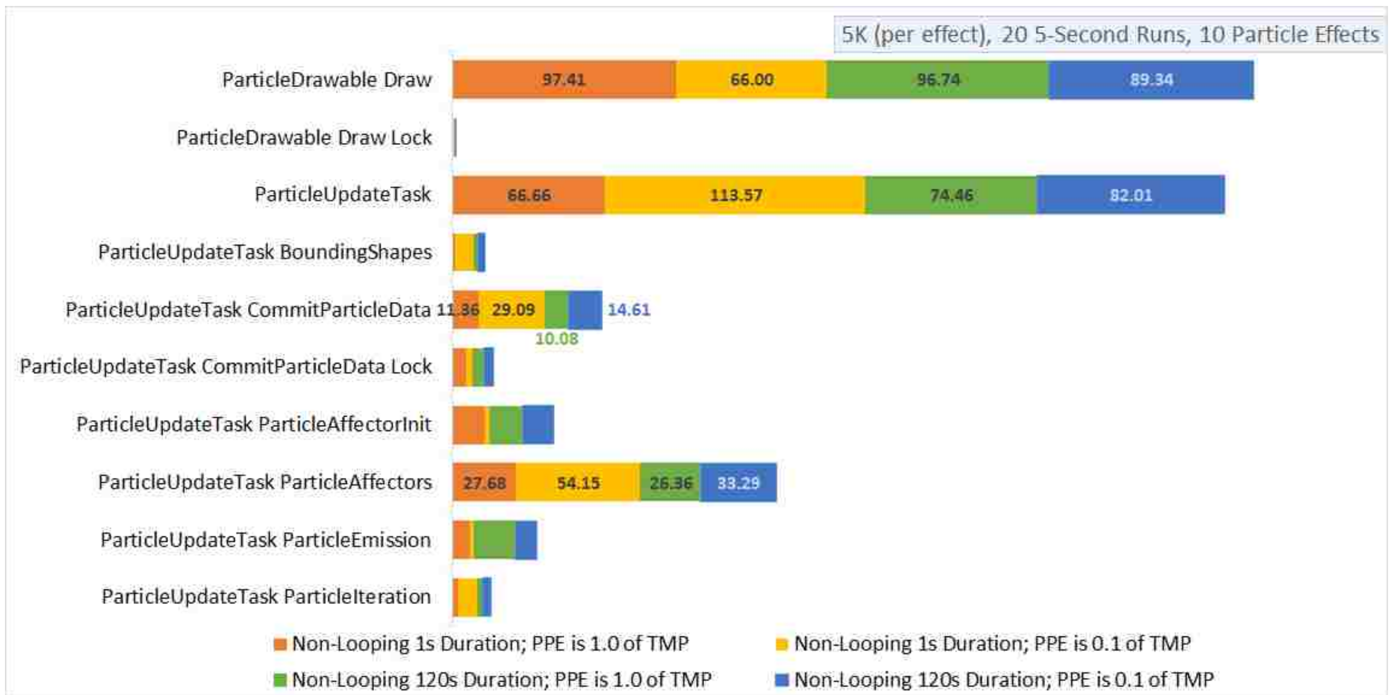

Figure 5.37 A graph showing the time spent in various sections of code for ten particle effects. This shows: (1) The multiple-burst (or 0.1 TMP) games for the Non-Looping 1s Duration data series has a higher run time for particle update functions and a lower run time for particle draw functions than the Non-Looping 120s Duration data series, and (2) The single-burst (or 1.0 TMP) games for the Non-Looping 1s Duration and Non-Looping 120s Duration data series both have approximately the same run time.

CHAPTER 6 CONCLUSION

The optimization case studies in Chapter 4 demonstrated the differences between managing contention for a single, large data structure and for many pieces or "shards" of a fragmented data structure. The developers that designed ChilliSource's *Particle Effect Components* created them such that "sharding" or breaking up the effect's particles array across many *Particle Effect Components* resulted in less contention between reader and writer threads due to the decreased chance that either thread would be contending for the same particle array. However, a solution that utilized "lock free" atomic read-modify-write operations could significantly reduce contention if a single, large data structure needed to be used. Although a "lock free" solution only improved ChilliSource's performance when there was a single particle effect, these findings are still useful for that scenario (i.e. two threads contending for a single, large data structure).

The observation studies in Chapter 5 explored three aspects of the particle effect: (1) the importance of the `ConcurrentParticleData` object, (2) the relationship between ChilliSource's *Task Scheduler* and *Particle Effect Components*, and (3) the perplexing scenarios in which particle effects do not visually emit particles.

The first study showed that using the `ConcurrentParticleData` object for a single particle effect shifts the heavy contention to itself instead of the *Particle Drawable*. In other words, if the `ConcurrentParticleData` object was not used during a game with a single particle effect attached to the ball, then the *Particle Drawable* would take on the contention that was observed within the `CommitParticleData` function which resulted in fewer particles being drawn to the screen. For 10 particle effects, however, there

was not any contention to begin with so using the `ConcurrentParticleData` object made little difference.

The second study demonstrated that, depending on how background tasks are organized, scheduling a background task may not necessarily be faster than its single-threaded implementation. ChilliSource's *`ParticleEffectComponent`* leverages the fact that threaded solutions generally do well if many threads work on many small problems at once; we have seen throughout this study that particles distributed across many *`Particle Effect Components`* (and thus many threads) consistently perform better than their single-threaded counterparts.

Unlike the other studies that found results that could be applied to software engineering as a whole, the third and final study investigated a problem that was limited to ChilliSource and its *`Particle Effect Component`*. I found that, when using a non-looping particle effect with a large number of particles, a large duration value should be set as well. Otherwise, particle effects may not visually emit. Although this was partially a misconfiguration on my part, it is something that ChilliSource may want to consider fixing so that a game developer would not have to experience the same confusion as I did.

Aspects of ChilliSource that I would consider for future studies would be ways to further optimize the *`Particle Effect Component`*. Although the optimization case studies improved contention in `CommitParticleData`, only the games that utilized one particle effect benefited since games with multiple particle effects did not experience contention in the first place. A possible avenue of optimization would be improving the performance time of *`Particle Affectors`*. Although I did not focus on the run time of *`Particle Affectors`* in this thesis, the runtime of *`Particle Affectors`* is significant enough to investigate. Any figures that listed timing metrics would demonstrate this, but a specific example of this can be found in Figure 4.19. If we consider the "Original; PPE is 0.1 of TMP" data series, then we can see that about 27 of 75 seconds of `ParticleUpdateTask`'s time is spent updating the effect's *`Particle Affectors`*. Since this is about 36% of its

time, it is worth a closer look.

APPENDIX A INSTRUMENTATION APPENDIX

In this appendix, the instrumentation application systems that I created are explained in great detail. This information is not critical to understanding my results, but the inner-workings of these complex systems are here for those who are interested.

A.1 Overview

Different parts of the engine were instrumented in order to numerically quantity what occurs during CSAPong games. Various metrics (e.g. particles actually emitted) were examined, and various sections of code were timed. These two use cases were separated into two application systems that I created that resided within the engine itself, the *Metrics* and *Timing* systems. As mentioned before, these systems will save a series of files containing the gathered information into a timestamped folder within CSAPong's save data file system. These files fueled the direction of these studies due to their ability to answer questions about the engine such as "What function in *Particle Effect Component* takes the longest to run?"

A.2 Metrics System

A.2.1 Metadata

Although not structurally complicated, the *Metrics System* keeps track of a great deal of primitives that describe the series of games run by CSAPong. The following is a list of the tracked primitives that remained constant throughout the games:

attribute	value
are particles looping	TRUE
num particle effects	1
min particles	500
max particles	2000
particles step	500
is total max particles changing	TRUE
is particles per emission changing	TRUE
particles per emission	0
total maximum particles	0
particles per emission step	50
total maximum particles step	500
total num runs	2
duration per run	5

Figure A.1 An example of metadata output by the `Metrics System`. The important part to understand here is that these games are fall under the multiple-burst scenario. In other words, the particle effect emits its particles over time. As Figure A.2 illustrates, the particles per emission (PPE) is always 10% of the effect's total maximum particles (TMP) because the step values (PPE step is 50, TMP step is 500) are what dictates the amount is added to PPE and TMP for each game as time goes on.

CSAPong Game Num	Particles Per Emission	Total Maximum Particles	Particles
1	50	500	500
2	100	1000	1000
3	150	1500	1500
4	200	2000	2000

Figure A.2 An example of the changing particle values during games in CSAPong. This shows a series of games that fall under the multiple-burst scenario with a min/max/step of 500/2000/500 and with particles per emission (PPE) emitting 10% of the effect's total maximum particles (TMP).

- ***Are particles looping***
  - Whether or not the particle effect is looping. This means that the particle effect will continue even after the duration has elapsed.
- ***Number of particle effects***
  - The number of `Particle Effect Components` that are attached to the ball entity.
- ***Is total maximum particles changing***
  - Whether or not *total maximum particles* is changing across different particle effect types.
- ***Total maximum particles*** (or TMP)
  - If *is total maximum particles changing* is false, then this will be the constant value of *total maximum particles* across particle effect types.

- ***Is particles per emission changing***
  - Whether or not *particles per emission* is changing across different particle effect types.
- ***Particles per emission*** (or PPE)
  - If *is particles per emission changing* is false, then this will be the constant value of *particles per emission* across particle effect types.
- ***Minimum particles***
  - The minimum value for the changing values (i.e. *total maximum particles* and/or *particles per emission*).
- ***Maximum particles***
  - The maximum value for the changing values (i.e. *total maximum particles* and/or *particles per emission*).
- ***Particles step***
  - The step value for the changing values (i.e. *total maximum particles* and/or *particles per emission*). This is used in conjunction with the minimum and maximum particles values to determine how many particle effects to create and use during CSAPong's execution.
- ***Particles per emission step***
  - If *is particles per emission changing* is true, then this will dictate the step value for *particles per emission* across particle effect types. This is generally a fraction of *particles step*.
- ***Total maximum particles step***
  - If *is total maximum particles changing* is true, then this will dictate the step value for *total maximum particles* across particle effect types. This is generally a fraction of *particles step*.
- ***Total number of runs***
  - The number of games to run per particle effect type.
- ***Duration per run***
  - The amount of time in seconds that a single game lasts.

Figure A.1 can be examined in order to make the distinction clear between the different particle variables. The *minimum particles* is 500, the *maximum particles* is 2000, and the *particles step* is 500. This tells us right away that there will be 2000/500 = 4 particle definition files that will be created and played[1]. Note that this is similar to the example shown near the end of Chapter 3 which also had a min/max/step of 500/2000/500.

From the figure, we also know that *particles per emission* and *total maximum particles* are changing over time and they will not be constant values. Thus, their ”step” values will tell us how much each variable (i.e. total maximum particles and particles per emission) will be added to as the game advances. We can also see that the *total maximum particles step* is 500, or 100% of the *particles step*, and the *particles per emission step* is 50, or 10% of the *particles step*. This is the same as the example variable blues given for the multiple-burst scenario in Chapter 3.

Figure A.2 shows how all of these particle variables change as CSAPong executes. Note that, since there are 4 different kinds of particle effects and 2 *total number of runs*, CSAPong would play a total of $4 \times 2 = 8$ times in this example.

A.2.2 Metrics

The `Metrics System` keeps track of three engine metrics, but it also manages six system metrics. Engine metrics are metrics that originate from the engine, e.g. counting the number of times that each particle was drawn. System metrics are metrics that come from the metrics system, e.g. the current game run number. In other words, system metrics come from metadata values, but they are printed with the engine metrics in order to give

run num	per effect total max particles	per effect particles per emission	all effects total max particles	all effects particles per emission	all effects particles actually emitted	all effects particles actually rendered	render ball called
0	500	50	500	50	500	49250	340
1	500	50	500	50	500	34250	302
0	1000	100	1000	100	1000	69400	300
1	1000	100	1000	100	1000	68500	302
0	1500	150	1500	150	1500	102300	300
1	1500	150	1500	150	1500	101700	302
0	2000	200	2000	200	2000	135600	300
1	2000	200	2000	200	2000	135000	302

Figure A.3: An example of metrics output by the `Metrics System` based on the metadata from Figure A.1.

[1]Note that we do not subtract 500 from 2000 since the range is inclusive.

them context. The following list shows all of the different engine and system metrics that are tracked:

- ***Engine Metrics***
  - ***All effects particles actually emitted***
    - As the game progresses, the size of the `New Particle Indices` array after each emission is added to this metric.
  - ***All effects particles actually rendered***
    - This is incremented every time a particle is rendered in the *`Drawable`* instance.
  - ***Render ball called***
    - This is incremented every time the ball is rendered.
- ***System Metrics***
  - ***Run number***
  - ***Per effect total max particles***
  - ***Per effect particles per emission***
  - ***All effects total max particles***
  - ***All effects particles per emission***

A.2.3 Particle Effect Definition

As mentioned in Chapter 2, *`Particle Effects`* that are passed into *`Particle Effect Components`* cannot be created programmatically. They must be created with JSON particle definition files. If tens or hundreds of particle effects needed to be created, then just as many particle definition files would have to be made. This is clearly something that should be done by hand, and so a script, `generate_particles.py`, was created to automate this process. An example invocation of this script is shown in Code Listing 23, and that invocation will generate the particle definition files shown in Figure A.5. Note that these generated particle effects are the same ones that were used in Figure A.1.

Code Listing 23: An example invocation of the generate particles script.

```
1 python generate_particles.py
2 > -changing Both
3 > -min 500 -max 2000 -step 500
4 > -tmpStep 1.0 -ppeStep 0.1;
```

```
$ python generate_particles.py -h
usage: generate_particles.py [-h] -changing {PPE,TMP,Both} [-min MIN]
                             [-max MAX] [-step STEP] [-constant CONSTANT]
                             [-tmpStep TMPSTEP] [-ppeStep PPESTEP] [-dir DIR]

Generate Particle JSON Files

optional arguments:
  -h, --help            show this help message and exit
  -changing {PPE,TMP,Both}
                        Either the particles per emission (PPE) or total max
                        particles (TMP) will be changing from min to max.
  -min MIN              The minimum particles- either PPE or TMP. (non-
                        negative number, default: 0)
  -max MAX              The maximum particles- either PPE or TMP. (non-
                        negative number, default: 1000)
  -step STEP            The step between min and max particles- either PPE or
                        TMP. (default: 100)
  -constant CONSTANT    The constant number of particles, either PPE or TMP
                        but it will be opposite of arg "changing". If both are
                        changing, then this has no effect. (default: 10000)
  -tmpStep TMPSTEP      The percent that TMP will step by if it is changing.
                        (default: 1.0)
  -ppeStep PPESTEP      The percent that PPE will step by if it is changing.
                        (default: 1.0)
  -dir DIR              The output directory name (default is "Generated" and
                        it will create it for you)
```

Figure A.4: Shows the parameters that the `generate_particles.py` script can use.

Name	Date modified	Type	Size
500_particles.csparticle	11/2/2016 9:39 PM	CSPARTICLE File	2 KB
1000_particles.csparticle	11/2/2016 9:39 PM	CSPARTICLE File	2 KB
1500_particles.csparticle	11/2/2016 9:39 PM	CSPARTICLE File	2 KB
2000_particles.csparticle	11/2/2016 9:39 PM	CSPARTICLE File	2 KB

Figure A.5: Shows the particle files that were generated by an invocation like in Code Listing 23.

A.3 Timing System

A.3.1 Visual Studio

ChilliSource generates a Visual Studio solution file for the developer to use when a project is created, and so Visual Studio was utilized as an IDE during development on a Windows machine. Visual Studio boasts a wide array of profiling tools which include (but is certainly not limited to) CPU sampling and function instrumentation. The CPU sampling was con-

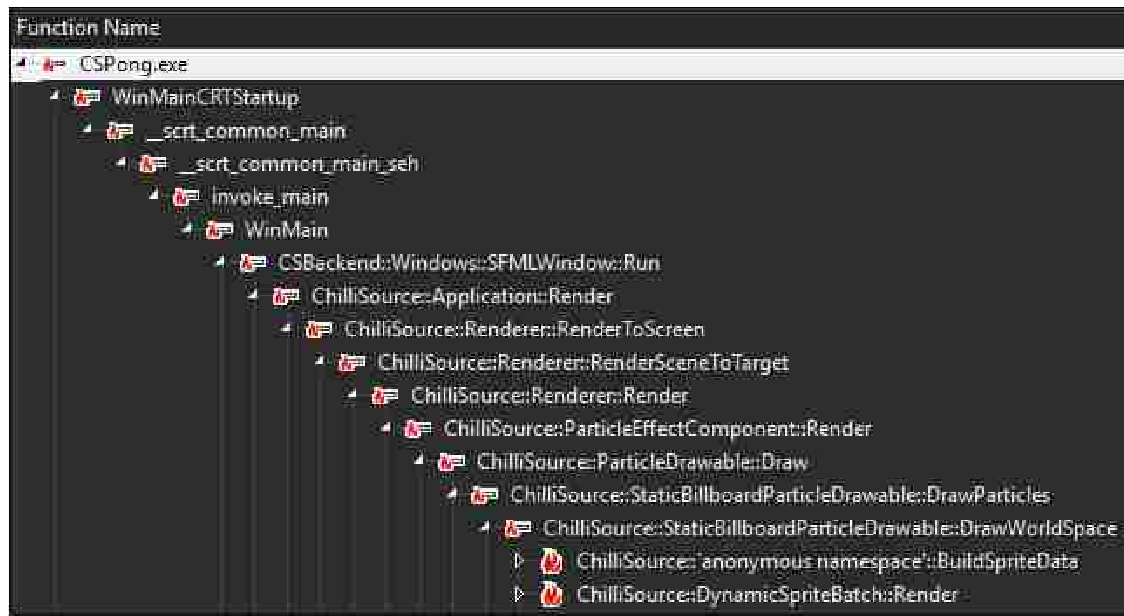

Figure A.6 Shows an example of Visual Studio's generated call tree for CPU instrumentation when ran using the metadata from Figure A.1. This call tree in particular shows the "hot path" of execution, or the path that leads to the "function leaf" with the highest exclusive samples %. In this instance, the function calls that led to `ParticleDrawable::DrawParticles` was the "hot path" of execution.

Level	Function Name	Inclusive Samples %	Exclusive Samples %
8	ChilliSource::Renderer::RenderSceneToTarget	65.38	0.01
9	ChilliSource::Renderer::Render	45.64	0
10	ChilliSource::ParticleEffectComponent::Render	29.28	0
11	ChilliSource::ParticleDrawable::Draw	29.26	0
12	ChilliSource::StaticBillboardParticleDrawable::DrawParticles	27.42	0
13	ChilliSource::StaticBillboardParticleDrawable::DrawWorldSpace	27.35	0.11
12	ChilliSource::TaskPool::ProcessTasks	19.32	0
13	ChilliSource::TaskPool::PerformTask	19.32	0
20	ChilliSource::`anonymous namespace'::ParticleUpdateTask	19.04	0.12
9	ChilliSource::Renderer::RenderShadowMap	13.02	0
10	ChilliSource::Renderer::RenderShadowMap	12.87	0.02
14	ChilliSource::`anonymous namespace'::BuildSpriteData	11.4	0.29
14	ChilliSource::DynamicSpriteBatch::Render	10.63	0.09
10	ChilliSource::StaticMeshComponent::Render	7.12	0
11	ChilliSource::Mesh::Render	7.11	0
21	ChilliSource::ColourOverLifetimeParticleAffector::AffectParticles	6.8	0.18
12	ChilliSource::SubMesh::Render	6.8	0

Figure A.7 Shows the same call tree from Figure A.6, but sorted and filtered. The function call tree level is ≥ 8, the function names begin with "ChilliSource::", the inclusive samples % is $\geq 6\%$, and it is sorted by the inclusive samples % in descending order. These parameters were used to ensure that the results were low-level ChilliSource function calls with a high inclusive sample percent. The functions with the most inclusive samples are either related to rendering or updating particles.

siderably helpful during the initial stages of these studies. The inclusive (includes samples from all functions calls within it) and exclusive (only includes samples from itself) sampling pointed out hot spots within the `Particle Effect Component` to investigate. The call trees that the sampling produced (see figures Figure A.6 and Figure A.7) also assisted in understanding how the engine worked. Although all of this information is helpful, the CPU sampling does not provide concrete times. The function instrumentation that Visual Studio provided should have filled that gap, but it was not used simply because it did not reliably and smoothly work. It would slow down the game considerably during execution, and then would take a great deal of time to process after execution. Even if it did work, it did not have the flexibility to instrument custom sections, and it also could not be used to instrument the game on other platforms such as iOS or Android.

A.3.2 Shiny

Seeking a way to instrument functions and custom blocks of code across all platforms, a third party library called Shiny was used. Shiny is an older C/C++/Lua intrusive profiler created by Aidin Abedi with ”very very low overhead” [10]. It allows the user to insert macros within functions and code blocks, and it outputs a call tree very similar to Visual Studio's with the time used by the named code blocks and function calls. Although Shiny's creator asserted that Shiny is ”amazingly simple to use and flexible”, it took about a week to integrate Shiny within CSAPong and ChilliSource in such a way that it worked on Windows, iOS, and Android. The majority of the encountered problems had more to do with compilation errors and flags, however, than the usage of Shiny's API. The API of Shiny is actually simple and flexible to use, as shown in Code Listing 24. Regrettably, Shiny has two major flaws. First, it does not reliably run if the program is multi-threaded (and ChilliSource is multi-threaded). Second, the output is not easily parsed since it is not separated by delimiters, and parsable output was required to quickly format, collate, and examine results during this thesis.

Code Listing 24: Using the pseudocode from Code Listing 5, this shows how *Shiny* could be used to instrument ParticleUpdateTask.

```
1  ParticleUpdateTask(copiedAtts)
2  {
3    // Use Shiny macro to profile the entire block
4    PROFILE_SHARED_BLOCK(ParticleUpdateTask);
5
6    // Use Shiny macros to profile outside and inside of the
7    // particle iteration loop
8    PROFILE_SHARED_BEGIN(ParticleIter_OuterLoop);
9    for(particle in copiedAtts.particleArray)
10   {
11     PROFILE_SHARED_BEGIN(ParticleIter_InnerLoop);
12     if(particle.isActive)
13       particle.UpdateValues();
14     PROFILE_END();
15   }
16    PROFILE_END();
17
18   // ... and so on
19 }
```

A.3.3 Timing Application System

I developed the *Timing System* due to dissatisfaction with the Visual Studio and Shiny function profilers. This system reliably instruments the multi-threaded engine across all platforms and returns results that are easily parsed. It uses the built-in *Performance Timer* from ChilliSource along with its own StartTimer and StopTimer static methods in order to time code blocks in a similar fashion to Shiny, as shown in Code Listing 25. To achieve this, it uses three hash tables[2] with the following key-value pairs:

- Timing Hash Table
  - KEY: (std::string) Code section name
  - VALUE: (double) Total time
- Timer Hash Table
  - KEY: (std::string) Timer key
  - VALUE: (ChilliSource::PerformanceTimer) Timer object
- Counting Hash Table

[2] In C++ 11, the closest thing to a hash table is an unordered map from the std namespace.

timed section name	time in seconds	num section calls
ParticleUpdateTask	5.74985	1220
ParticleIter_OuterLoop	0.298633	1220

Figure A.8: An example of timing output by the *Timing System* with just the ParticleUpdateTask and the *Particle Iteration* times.

- KEY: (std::string) Code section name
- VALUE: (unsigned int) Number of code section calls

Code Listing 25: Using the pseudocode from Code Listing 5, this shows how our custom *Timing System* could be used to instrument ParticleUpdateTask.

```
ParticleUpdateTask(copiedAtts)
{
  // Statically call the timing system to start a timer with hash key
  // "ParticleUpdateTask".
  pUpdateTimerKey = TimingSystem::Start("ParticleUpdateTask");

  // Use the timing system to keep track of the outer loop times
  // during particle iteration.
  pOutIterTimerKey = TimingSystem::Start("ParticleIter_OuterLoop");
  for(particle in copiedAtts.particleArray)
  {
    if(particle.isActive)
      particle.UpdateValues();
  }
  TimingSystem::Stop("ParticleIter_OuterLoop", pOutIterTimerKey);

  // ... and so on

  // Stop the timer with hash key "ParticleUpdateTask" and the outputted
  // timer key
  TimingSystem::Stop("ParticleUpdateTask", pUpdateTimerKey);
}
```

A.4 Output and its Evolution

From the examples that were shown above, it can be observed that the system output lends itself to a tabular structure that follows a comma-separated-values format. Storing

the data like this was simple, but storing the metadata proved to be more difficult.

At first, the metadata was stored all within the titles of the CSV (e.g. metrics_maxRunNum = 5_particle-effects = 1 ... [timestamp].csv). This worked well when there were only a couple of metadata values, but, as shown earlier in this chapter, there are now a great deal of metadata values. Predictably, the titles of the CSVs eventually became so unruly that Windows refused to open the files because the "file path was too long". This problem was solved by first bundling the metrics and timing CSVs into one timestamped directory, and then outputting another CSV file exclusively for metadata in that same directory.

Although the metadata problem was solved, bundles of CSVs were not a particularly tidy approach when it came to distribution. However, any other method of outputting the data in C++ would be far too complicated and out of scope for the purposes of our studies. Outputting the data in another format using python, however, would not be as complex. I created a python script to import the three bundled CSVs into an HDF5 file. HDF5, i.e. Hierarchical Data Format 5, is "a unique open source technology suite for managing data collections of all sizes and complexity," [11], designed to support large, complex datasets that could be used on every size and type of system. HDF5 is hierarchical, high-performance, portable, can be used in numerous languages, and it is self-describing. Admittedly, HDF5 may be overkill for my tens of CSVs, but its ability to efficiently bundle all of the output files with metadata built right in is still useful. The HDF5 file that contains all of the results presented in this thesis can be found in a Bitbucket repository [6].

BIBLIOGRAPHY

[1] W. T. Reeves, "Particle systems – a technique for modeling a class of fuzzy objects," in *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '83. New York, NY, USA: ACM, 1983, pp. 359–375. [Online]. Available: http://doi.acm.org/10.1145/800059.801167

[2] ChilliWorks. (2014) Features. [Online]. Available: http://www.chillisourceengine.com/features

[3] ——. (2014) Life cycle events. [Online]. Available: http://www.chillisourceengine.com/life-cycle-events

[4] ChilliWorks and A. Gross. (2016) Chillisource automated cspong game. [Online]. Available: https://bitbucket.org/angelahnicole/um-thesis-cspong-benchmarking/src/c00b947411a7e04e00d464108dee6235e3d692c1/?at=automation

[5] ChilliWorks. (2016) Chillisource sample projects. [Online]. Available: https://github.com/ChilliWorks/CSSamples

[6] A. Gross. (2016) Chillisource game engine particle system study dataset. [Online]. Available: https://bitbucket.org/angelahnicole/um-thesis-particle-optimization/downloads/csgeps_angela-gross_thesis_data.hdf5

[7] W. Goesgens. (2015) Comparison: Lockless programming with atomics in c++ 11 vs. mutex and rw-locks. [Online]. Available: https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/

[8] S. Mullender and R. Cox, "Semaphores in plan 9," in *3rd International Workshop on Plan*, vol. 9, 2008, pp. 53–61.

[9] cppreference. (2015) Atomic compare exchange. [Online]. Available: http://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange

[10] A. Abedi and D. Love. (2011) Package of the shiny profiler by aidin abedi. [Online]. Available: https://github.com/dlove24/Shiny

[11] HDFGroup. (2016) High level introduction to hdf5. [Online]. Available: https://support.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf

[12] ChilliWorks. (2014) Basic structure. [Online]. Available: http://www.chillisourceengine.com/basic-structure

[13] Halixi72. (2007) Particle emitter. [Online]. Available: https://en.wikipedia.org/wiki/File:Particle_Emitter.jpg

[14] ——. (2007) Strand emitter. [Online]. Available: https://en.wikipedia.org/wiki/File:Strand_Emitter.jpg

[15] ChilliWorks and A. Gross. (2016) Chillisource v1.6.0 code repository. [Online]. Available: https://bitbucket.org/Chilli-Ian/chillisource-ag/src/781988bb9a8ee518c4ea6545ac0e5c55238ec562?at=temp/particleMetrics