Performance of Priority-Based Game Object Scheduling

NATHAN BHAT

FILIP ÅSBERG
Performance of Priority-Based Game Object Scheduling

NATHAN BHAT & FILIP ÅSBERG

Date: June 2, 2018
Supervisor: Christopher Peters
Examiner: Örjan Ekeberg
Swedish title: Prestanda av prioriteringsbaserad schemaläggning av spelobjekt
School of Electrical Engineering and Computer Science
Abstract

This paper investigates alternative scheduling for game objects in a game engine by using priority based algorithms and comparing different priority assigning methods. Research concerning commercial game engines, game loops, concurrency and scheduling was performed. Three different scheduling algorithms were created and tested using a varying amount of game objects between two different testing scenarios. The results showed that using priority based scheduling gave a slight increase in average frames per second (FPS) as well as boosting logical update performance of prioritized objects, however overall performance was decreased. The reason this decrease in performance occurred could have been due to the way the scheduler was implemented, and despite the decrease in performance using a priority based scheduling system was considered feasible within certain game engines.

Sammanfattning

# Contents

1 Introduction ................................................. 6  
1.1 Scientific relevance ........................................ 6  
1.2 Keywords .................................................. 7  
1.3 Research questions ........................................... 7  
1.3.1 Problem Statement ........................................ 7  
1.3.2 Questions ................................................. 7  

2 Background .................................................. 9  
2.1 Graphic Engines .............................................. 9  
2.1.1 Case study: OGRE ......................................... 9  
2.2 Game Engines ............................................... 9  
2.2.1 Case study: Game Maker Studio ......................... 13  
2.2.2 Case study: Unity ....................................... 14  
2.2.3 Case study: Unreal Engine .............................. 15  
2.2.4 Case study: Godot ....................................... 16  
2.2.5 Game Objects .......................................... 16  
2.3 Game loop .................................................. 17  
2.4 Concurrency ............................................... 19  
2.5 Scheduling ............................................... 21  
2.6 Other related work ........................................... 21  

3 Method ..................................................... 23  
3.1 Research ................................................ 23  
3.2 Evaluation ............................................... 23  
3.3 Implementation ........................................... 25  

4 Results .................................................... 28  
4.1 Scenario 1 ............................................... 28  
4.1.1 Sequential scheduler .................................... 28  
4.1.2 Priority scheduler ...................................... 30  
4.2 Scenario 2 ............................................... 31  
4.2.1 Sequential scheduler .................................... 31  
4.2.2 Priority scheduler ...................................... 32  
4.2.3 Angle scheduler ...................................... 34
5 Discussion

5.1 Scheduler results ........................................ 37
5.2 Comparison between schedulers .......................... 38
  5.2.1 Sequential vs. Priority ................................. 38
  5.2.2 Sequential vs. Angle .................................. 39
  5.2.3 Priority vs. Angle .................................... 39
5.3 Accuracy of results ....................................... 40
5.4 Feasibility .................................................. 41
5.5 Future work ............................................... 42

6 Conclusion .................................................... 43

Appendices ....................................................... 44

A Glossary ....................................................... 44

B Highlighted Pseudocode .................................... 45
1 Introduction

Commonly during game development a game engine is used. A game engine can be seen as a library containing functionality needed for game creation. Game engines often offer functionality that simplifies rendering and processing game logic. A frequently defined element within a game engine is a game object. This object, in most cases, contains several types of components that the programmer can use to make the object behave as intended. These components can include physics behavior, collision behavior, rendering, scripts, artificial intelligence (AI), etc. An example of a game object is a player-controlled character.

Game objects are typically updated evenly, that is to say: if one game object is updated, then so is every other game object that is active. While this approach helps induce deterministic behavior, it may seem unreasonable for all game objects to be updated equally, for example if an object that is purely aesthetic and in the background is updated as frequently as an object that is vital for the gameplay. Therefore this paper aims to study possible alternatives to the traditional updating method, to attempt prioritizing objects deemed important. This is intended to increase performance in user defined scripts, primarily AI and player interaction within prioritized objects.

Furthermore, some amount of concurrency will be introduced between game objects’ updated calls. This is to ensure responsiveness, despite introducing an unfair scheduling algorithm as more objects can be scheduled if each object uses a smaller amount of processor time.

1.1 Scientific relevance

Researching new and developing methods within game engine technology is relevant as the computer games industry is rapidly expanding and game engines are directly linked to the software aspect of game development.

Rendering complex graphics models with many objects doing individual logic has become increasingly important within computer science. A lot of scientific research uses graphical models and simulations to prototype, experiment and develop, and in many cases these models need to handle complex imagery and simulation. The leading tools to perform these simulations within real world applications are game engines (Hu et al. 2014).

The types of simulations relevant to this method of testing are simulations that need to graphically visualize many objects in context and simulate
interaction between these objects; that is to say not in the form of tables and charts, but in a virtual world setting. According to (Uskov & Sekar 2014) large companies such as Google, Microsoft, IBM, Oracle, among others, have started games and gamified applications to support their activities in simulation of physical objects, project management, engineering etc.

Not only are game engine applications useful for engineering, they can also be used in other research areas. Simulation is becoming an increasingly important research method, and (Marks et al. 2007) highlights that:

“Game engines offer unique advantages for the creation of highly interactive and collaborative environments.”

and proceeds to explain that game engines can be used for creation of simulations for medical education and surgical training.

Since this paper studies some aspects of game engine performance, along with the information given in this section, the scientific relevance of this study is apparent.

1.2 Keywords

Game engine, game loop, game object, concurrency, scheduling, coroutine, update calls, gameplay, prioritizing.

1.3 Research questions

1.3.1 Problem Statement

In game engine logic, objects are updated sequentially and evenly, while certain objects may be more important to the player experience than others. Sequential game loop programming may have a potential performance increase, feasible by introducing priority-based updating of game objects.

The term “performance” is defined in this case as the amount of times an object is updated. If a game object is updated more frequently, internal game logic such as AI will iterate an increased amount of times. This type of performance is often directly linked to how accurately a game object behaves.

1.3.2 Questions

- Does adding priority-based scheduling for game objects in the commercial Game Engine Godot increase, or decrease, performance of prioritized game objects and frames per second?
This research question is meant to create a discussion about the difference between amount of times updated for high priority game objects and low priority game objects. It is also meant to create a discussion about the feasibility of using a priority-based scheduling system in a commercial game engine. If using a priority-based scheduler gives an increase in performance of prioritized game objects, without lowering the performance of low priority objects to unreasonably low levels (levels that would affect gameplay experience), then using a scheduling system like this may be a feasible feature applicable to a game engine. However, if the performance is decreased, due to the complexity of the scheduling algorithm, this scheduling system may not be applicable in a game engine.

- *Does using a priority-based scheduler in the commercial Game Engine Godot affect system performance?*

This research question raises the discussion of how different algorithms for prioritizing game objects affects the overall performance, as in performance regarding both low priority and high priority objects. Some scheduling algorithms may be complex, making scheduling a heavy process. Depending on how the different scheduling algorithms perform this question will answer whether or not these algorithms are feasible as features in a commercial game engine.
2 Background

2.1 Graphic Engines

According to the Linux Information Project, the definition of a graphic engine is the following:

“A graphic engine is software that is used by application programs to draw graphics on computer display screens.” (Graphic Engine Definition 2005).

2.1.1 Case study: OGRE

An example of a graphics engine is OGRE (Acronym for: Object-Oriented Graphics Rendering Engine). More specifically, OGRE is a scene-oriented, 3D engine, meaning that it can be used to render scenes containing 3D objects, according to OGRE’s “About” page (Object-Oriented Graphics Rendering Engine 2018). Being able to produce and render 3-dimensional scenes is an integral part of any game engine, however there are some distinct differences separating OGRE from being a game engine. Being a graphics engine, OGRE does not account for libraries that might be essential for a game engine, such as: sound, networking, input and collision (Object-Oriented Graphics Rendering Engine 2018, Getting Started, OGRE wiki).

2.2 Game Engines

A game engine is a piece of software with reusable components, used to develop a software-based game.

The structure of a game engine can vary depending on what genre and type of game that is being created, according to (L. Bishop 1998, p. 1). However, Bishop et al. define a game engine as having the following parts: Input, Audio, Graphics, Dynamics (dynamic data such as: physics, collision etc.) and an event loop, each of which handle their respective components in the game (L. Bishop 1998, p. 2). This is explained in Figure 1.
The separation between game and game engine is defined differently depending on the game. Some render components may have unique methods to render data specific to the game (such as a character), while other systems may define the line between engine and game clearer by using general purpose methods for rendering (Gregory 2009, p. 11). Gregory argues that because of this the ability to reuse game engines varies depending on the engine, and that the more general purpose the engine is, “the less optimal it is for running a particular game on a particular platform” (Gregory 2009, p. 12).
Figure 2: A schematic explaining a general purpose game engine using a message bus. The “Game Logic” element is specific to the game, not the engine. (Kissner 2015)

This overview of a game engine is however abstracted, and in practice it might be more useful to go into more technical detail, as seen in Figure 3.
There are an array of different commercial and public game engines today. Some of these game engines have been described.
2.2.1 Case study: Game Maker Studio

![Game Maker Studio](image)

Figure 4: Game maker studio. A general purpose 2D game engine (*Game Maker Studio 2018*)

Game Maker Studio, seen in Figure 4, is an engine primarily used for 2D games, however, with some effort, can be used for 3D games as well. It utilizes visual programming and is built on a language developed for the engine, GML (*Game Maker Studio 2018*).
2.2.2 Case study: Unity

In Figure 5, the GUI of the Unity Game engine is described. Unity is a widely used engine, that supports both 2D and 3D game programming. The engine supports scripting in both C# and UnityScript (a language made for unity, modelled after JavaScript). The engine has a lot of powerful features and is often used for mobile game development (Unity 2018).
2.2.3 Case study: Unreal Engine

Unreal Engine, seen in Figure 6, is a powerful 3D game engine. It is widely used in the industry, and supports a wide variety of cutting edge features. The engine supports visual scripting and scripting in C++ (Unreal Engine 2018).
2.2.4 Case study: Godot

Figure 7: Godot. A open source, free, 2D and 3D game engine (Godot 2018)

Figure 7 shows the Godot game engine, a free and open source game engine used for both 2D and 3D games. It supports scripting in both C++ and C#, however some extra prerequisites are needed to develop using C#. Godot also supports GDScript which is made for the engine and likens Python, visual scripting and further language support. Being open source, it means that the engine’s code is readily available for analysis and reuse, making it transparent compared to the previous engines. However since the engine is made by volunteers, the software is not as cutting edge as other commercial engines.

2.2.5 Game Objects

Game objects (“Game objects” may be a title specific to the engine; see Appendix A for more titles) are Actors in a Game engine that serve a purpose (Equis 2010). More specifically, a Game Object may contain several components such as physics behavior, rendering behavior, collision, position, user defined scripts and so on. Game Objects do not have any set definition, and in some cases it can be difficult to know where one game object ends and where another one starts (Xiaoyu Ge & Zhang 2016). However in most cases
a game object will only contain one sprite or mesh, and all of the behavior related to that sprite or mesh (Amarasinghe & Parberry 2013). Examples of common game objects are player characters, walls, lights, obstacles and non-player controlled characters.

2.3 Game loop

A game loop is the core programming concept on which most games are built. (Nystrom 2014, p. 123-138) mentions that game loops were one of the earliest forms of creating an interactive program; one that could give immediate response and change depending on user input.

(Joselli et al. 2009, p. 2) explains that Real-time systems are solutions that have constraints regarding execution times and that games are examples of these systems. Game loops can be implemented in these systems to reach these constrains.

At a basic level, game loops has three tasks. It processes user input that has occurred since the last iteration of the loop (data acquisition), it updates the logic of each object in the game, such as position or state, (data processing) and it renders the game on screen (data presentation) (Joselli et al. 2009, p. 2). Since it is advantageous to run a game loop at a set amount of times per second, to ensure a game that runs at a constant speed, it is common to let the loop sleep a few milliseconds. (Nystrom 2014, p. 123-138) has shown a simple implementation of this concept in the following figure and pseudocode:

![Figure 8: A schematic explaining a simple game loop with time correction (Nystrom 2014)](image)
Listing 1: Game loop with time correction (Nystrom 2014)

\[
\text{while (true)}
\{
\text{double start = getCurrentTime();}
\text{processInput();}
\text{update();}
\text{render();}

\text{sleep(start + MS\_PER\_FRAME - getCurrentTime());}
\}
\]

Nystrom explains that if the loop can execute in less time than \text{MS\_PER\_FRAME}, then the \text{sleep()} call will halt the game loop, so the game does not speed up. However Nystrom elaborates that the game will slow down if the game loop takes longer than the given time to execute. Some of the suggestions Nystrom proposes to work around this fault is to simplify the graphics and the AI in the running application, cutting time from both the \text{update()} call and the \text{render()} call. Nystrom continues to suggest improvements to the game loop until it appears as the code below:

Listing 2: Game loop with lag correction (Nystrom 2014)

\[
\text{double previous = getCurrentTime();}
\text{double lag = 0.0;}
\text{while (true)}
\{
\text{double current = getCurrentTime();}
\text{double elapsed = current - previous;}
\text{previous = current;}
\text{lag += elapsed;}

\text{processInput();}

\text{while (lag >= MS\_PER\_UPDATE)}
\{
\text{update();}
\text{lag -= MS\_PER\_UPDATE;}
\}

\text{render(lag / MS\_PER\_UPDATE);}
\]
This idea is visualized Figure 9.

Figure 9: A schematic explaining a game loop with lag correction (Nystrom 2014)

In this version of the game loop, the system keeps track of how much time has passed since the last update and can run as many `update()` calls as should have been run during that time. Nystrom explains that this allows for two systems that perform differently to update correctly, without differing between systems when it comes to error rounding (when doing arithmetic with floating point values). This also allows for the `render()` call to render objects based on their velocities, their positions, and the `lag` value to know where each object should be. With this type of game loop, Nystrom argues that on a system that is not as powerful as another (when it comes to CPU and GPU performance), then the difference will be in frames per second, rather than logical updates (Nystrom 2014, p. 123-138).

### 2.4 Concurrency

Raynal describes a concurrent program or concurrent algorithm as “the description of a set of sequential state machines that cooperate through a communication medium, e.g., a shared memory”. This explains the idea of having a program execute several processes concurrently. These processes do not work independently of each other, but interact by manipulating the processes’ shared memory. When the number of processes of a program exceed
In some cases, because of some consistency reason, code can’t be executed in several processes at the same time. An example of this is when the same shared data is accessed by different processes, which can result in unexpected behaviour within the program (race conditions). In order to run the, as Raynal describes them, “critical sections” of the code on different processes, but not at the same time, the different processes can try to acquire a lock before executing a critical section of the code (Raynal 2012). Only one process can hold the lock while executing, and when the process is done executing it will release the lock. If a process tries to get the lock, and another process holds it - it will wait until the lock is released. By implementing a lock of this type, race conditions can be prevented (David A. Patterson 2012, p. 638).

Beerel et al. explain another common problem with concurrent programming: deadlocks. A deadlock is a situation where multiple processes are waiting for each other before they can complete their task, thereby mutually blocking each other from making progress (Beerel 2010, p. 106). Beerel et al. gives a formal definition of deadlocks as:
A set $P$ of processes is said to be deadlocked if:

- each process $p_i \in P$ is waiting for an event $e \in E$,
- each event $e \in E$ is generated only by a process $p_j \in P$.

Game engines often use concurrency in their underlying architecture, however using concurrency when programming game objects and in game AI is also possible, and commonplace. Jean Simonet, a programmer at Bethesda, has described in an article using concurrency to improve the sequential state machines that are in a game. As an example, he demonstrated, and provided the code for, an AI (artificial intelligence) that handled a turret in a game, that looked for and fired at the player; using coroutines, concurrency, instead of sequential logic (Simonet 2016a). In a related article, Simonet explains that classic sequential state machines are often over-engineered and overly complex even when solving a simple in-game task. Using a concurrent programming pattern to lighten the load of complexity is suggested by Simonet as an alternative method (Simonet 2016b).
Coroutines are a form of concurrency similar to the idea of subroutines. A subroutine is simply a helper function to a larger method, however a coroutine works independently and instead cooperates with other coroutines to create a pipeline. Coroutines build on the idea of yielding, to give the process back to another part of the code. Yield calls are explicit when programming with coroutines, however yields can often be programmed to return to the code after a set amount of time, or after an event occurs (DeNero 2014, ch. 5.3).

2.5 Scheduling

Several processes can run at the same time on a single CPU within an operating system. The abstraction of making these processes seem like they are running on separate CPUs is done by a scheduling policy within the operating system. A scheduling policy is what decides when to stop running a certain process, and start running another - called a context switch (Remzi H. Arpaci-Dusseau 2014, ch. 4, p. 2).

A multi-threaded program has more than one point of execution and consists of a process containing several threads. A thread can be described as an abstraction of a single running process. Like process scheduling, the operating system’s scheduling policy also decides when to run what thread within a process. Arpaci-Dusseau mentions that it may be hard to predict what the scheduler will run at a given point in time (Remzi H. Arpaci-Dusseau 2014, ch. 26, p. 4).

Within processes using concurrency, an internal scheduling policy can be applied. This scheduling does not have anything to do with the OS and is usually defined by the programming language or program itself.

2.6 Other related work

During the research period of this study, scientific research about game object scheduling was not found. Therefore there is no related work in this section that is directly coupled to this study. However studies were found with certain similar aspects to this paper, and have therefore been presented.

(Joselli et al. 2008) presents an architecture for physics engines, where rigid body simulation is the main focus and certain engine methods are implemented on the GPU. Thus, the physics engine distributes processes between the CPU and GPU, which unloads the CPU and allows it to process other tasks. The paper also presents an automatic process distribution
scheme between the GPU and CPU. The authors argue that the importance of the automatic distribution is derived from the fact that the simulated scene characteristics may change during the simulation. With the automatic distribution, the system can gain the best performance of both the CPU and GPU. The paper also demonstrates an uncoupled multithreaded game loop used by the physics engine. Based on the results presented in the paper, the authors concludes that distributing some physics calculation to the GPU is important because of two reasons. These two reasons are the following: unloading the CPU and optimizing the physics engine in order to have a greater amount of rigid bodies in the simulation.

(Joselli et al. 2009) presents an architecture used to implement all game loop models for real-time applications and games that use the GPU for co-processing mathematics and physics, working in parallel with the CPU. The model presented in the paper employs concepts of automatic task distribution. The architecture proposed in the paper acquires knowledge about the hardware by studying the performance over time of tasks running on each processor. By doing this, the architecture can find the best processor for a set of tasks.
3 Method

The following methodology was used to conduct a study, creating an implementation based on literature studies, to answer the research questions in section 1.

3.1 Research

The research aspect of this study began with acquiring knowledge about game engines. Articles were found using Google scholar and KTH’s online library that has access to ACM and ISO articles. The game development forum Gamasutra was also used to find information. Most articles and books used had abstracts (and abstracts of sections) that were used to determine the relevance of the article or book. We mainly searched for papers about game engines, game loops and game objects. There were not many studies similar to this one, creating the need to find different sources that together contained information relevant to this study. After a baseline of information about game engines were established, a game engine to perform the study with was able to be selected: Godot. We then researched game loops, concurrency and scheduling within game engines.

3.2 Evaluation

As discussed in Related work (section 2.6), (Joselli et al. 2009) presents an adaptive game loop architecture. This architecture uses both the GPU and CPU to process data and includes an automatic system for distributing tasks between them. While Joselli et. al’s paper is not directly coupled to this research, the evaluation can be considered similar as both these papers analyze performance. Based on the fact that Joselli et. al’s evaluation revolves around testing and results analysis, we chose to use testing and results analysis as the main form of evaluation for this study.

To create tests that help answer the research questions, the game engine Godot was used to implement three different schedulers, one baseline sequential scheduler, one using manual prioritization and one using automatic prioritization.

These tests included two different scenarios, meant to reflect different gameplay situations. For both scenarios, each scheduler was tested, using an increasing amount of game objects every test. The scenarios used two
different game objects, one that only had aesthetic purposes and one that contained a collision component and physics to simulate a more important game object. In the first scenario, 90% of the game objects were aesthetic and the rest were collider objects, whereas in scenario 2 this ratio varied slightly because of implementation purposes (variance never exceeded 1%).

Scenario 1 was set up so that the view frustum was static and could view all of the game objects simultaneously from above at an angle of 45 degrees. The game objects were generated in a rectangle shape, and the rows closest to the viewing frustum were the generated collider objects, while the rest were aesthetic. Scenario 2 was similar to Scenario 1 when it comes to the generation of game objects, however the view frustum was placed in the middle of the area of the game objects and rotated one full rotation during each test. The game objects in Scenario 2 were always generated in a square shape, to ensure an even distribution. The view frustum was pointed away from the collider objects when each test started and ended.

Scenario 1 and 2 were created to test different base cases. Scenario 1 represented a gameplay situation where the player would have full view of the game area, whereas scenario 2 was closer to a first person game, where the player is a perspective among the game objects. These two different scenarios were used because they give a wider representation of different game genres.

As previously mentioned, 3 different scheduling algorithms were used in the tests. The first scheduler was used as a base line as it did not use any priority, but instead scheduled the game objects sequentially. The second scheduler used a priority system, where each game object could be assigned a priority by the programmer (or designer, for that matter). The third scheduler used one of Godot’s built in functions to give a high priority to objects in front of the camera object, while giving low priority to objects behind the camera. Since Scenario 1 included all of the objects in front of the camera, the third scheduler, named the Angle scheduler, was excluded from that scenario.

For each scheduler six different tests were made, varying the amount of game objects. Each test measured, during the span of 5 seconds: the average times both aesthetic and collider objects were scheduled per object of the corresponding type, the average amount of times each collider object completed an entire update loop and the average frames per second (FPS) for the simulation. Each test was conducted three times, and the average of the three results was used in the Results section rounded off to the closest whole number (except for the average FPS).
3.3 Implementation

The main focus of this report is to test how well a priority scheduled game system would work. Since there are different choices to be made, such as how objects get prioritized, different scheduling approaches naturally emerge. We created 3 different schedulers to get a variety of solutions. The first scheduler was a base line and updated game objects sequentially without any prioritizing system. The second used manual prioritizing defined by the programmers. The final scheduler used an automatic prioritizing system, based on the angle and position of the camera object within the game engine used.

For the implementation Godot version 3.02 for windows was used. All of the tests were conducted on a computer with the following technical specifications:

- Processor: Intel(R) Core(TM) i7-6700k CPU @ 4.00GHz
- RAM: 16 GB DDR4
- GPU: GeForce GTX 970

The simulation starts with the system object, titled “MainSys”. When the simulation starts the system object spawns all of the game objects and starts recording data. Within Godot’s update call, called “process”, the scheduling algorithm is implemented. It iterates over the game objects and checks whether or not the game object should be scheduled based on its priority. The way the priority works, is the game object holds an internal integer representing its priority (a greater number means lower priority) and the scheduler skips scheduling the game object a number of times equal to the object’s priority before scheduling it. After a set amount of time, 17 milliseconds (about one sixtieth of a second), the system object stops scheduling objects, and lets Godot regain control to allow rendering etc. Psuedocode for the scheduler can be seen in Listings 3 in Appendix B.

The three schedulers were implemented in the following ways: the sequential scheduler always scheduled the game object, regardless of priority. The priority scheduler looked at the game objects internal priority which was an unchanging value and scheduled accordingly. For the angle scheduler there were two different priority values: a low priority value and a high priority value. Using Godot’s built in function for the camera object,
“is_position_behind”, the game object was assigned a high or a low priority, making it so that objects viewed by the camera, or close to being viewed, would have a higher priority. This is described in pseudocode in Appendix B, under Listings 4.

Because we needed specific functionality of all game objects we created a general game object that mimicked an interface. This general game object kept track of the game objects priority and whether or not it should be scheduled, and mainly served as a communication method between the system object and the actual game object. The general game object also contained a method for keeping track of how much time had passed since its child object was last in a specific section of code. This is necessary as the game object logic does not get to complete a full update before moving on to the next object, due to how the concurrency was implemented.

Each of the general game objects contained specific data, making it a specific game object. The custom update method that contained information about how the object behaved was stored here. Aesthetic objects were implemented using only a mesh instance, while collider objects used a rigidbody, a mesh instance and a collision shape.

The way this custom update method worked alongside the scheduler was that the scheduler had a coroutine for each object and started the coroutine when an object was scheduled. The object’s custom update would then explicitly call a yield to give the process back to the scheduler. Some concurrency was also implemented, so the collider object could do different parts of its custom update at different times.

To further explain how the collider objects and aesthetic objects differ, how the concurrency was implemented needs to be explained. To ensure a responsive scheduling practice, as responsiveness is important for interactive mediums, such as games, the custom update is implemented as a loop that contains several yield calls, that give the process back to the system object, so that other objects can be scheduled. In other words, an object that has more than one yield call would not get to complete an entire update loop before the next object was scheduled. This was implemented to increase responsiveness, however adding concurrency was not vital for this research.

The aesthetic objects and the collider objects differed mainly in their implemented custom update methods. While the aesthetic objects only contained one yield call, the collider objects contained three. This made it so that each time an aesthetic object was updated, it would complete its updated method before releasing the process. This was because the aesthetic
object only did one thing, which is rotate. The collider objects, however, rotated and “jumped”, as well as performed an arbitrary check to fill it out with more yield calls. This was done to test a more realistic behavior, as a real game object can do a lot in a single update call.

When the camera object was not static, that is to say in scenario 2, then it rotated around the y-axis at the speed of one rotation per five seconds. This was done to simulate a player in a first person setting, to get more accurate data when recording the angle scheduler.

As mentioned in the *Evaluation methodology*, the first scenario spawned game objects in a rectangular shape, while the second scenario spawned them in a square around the origin point. When the simulation started, a timer was started, and the amount of times each game object was scheduled and how many times it completed a full update loop was recorded. After 5 seconds this data was averaged between the different objects of the same type and printed to the console. At this time a new timer was started and ran for 5 seconds, recording the FPS. The average FPS for these five seconds was then printed to the console. We started the FPS recording 5 seconds after the simulation had started because at the start of the simulation there was always a dip in the FPS, which would lower the average.

Further details regarding the tests include that in Scenario 1 the camera object was placed so that all object were within the view frustum. In scenario 2 the camera object was placed slightly above the origin, so that it would have an even view of all objects when spun around. Then regarding priority: for the priority scheduler collider objects had priority 0, whereas aesthetic objects had priority 12 and in the angle scheduler high priority was set as 0 and low priority was set as 7.
4 Results

After the tests discussed in the previous method section, the results were recorded. This section presents those results.

Some abbreviations have been used when presenting data:

- No. of GO - Total number of game objects using the implemented scheduling system.
- Col. GO (%) - Percentage of game objects that are collider objects (see Chapter 3, Method).
- Avg. sch. - Average amount of times a game object was scheduled (only in tests where game objects were scheduled equally).
- Avg. aes. sch. - Average amount of times an aesthetic game object was scheduled during the test.
- Avg. col. sch. - Average amount of times a collider game object was scheduled during the test.
- Avg. col. lpd. - Average amount of times a collider game object completed an update loop during the test.
- Avg. FPS - Average Frames Per Second during the test.

All data was recorded for the duration of 5 seconds per test (FPS measurements started 5 seconds in to the simulation and lasted for 5 seconds).

4.1 Scenario 1

4.1.1 Sequential scheduler

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. sch.</th>
<th>Avg. col. lpd.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>9,712</td>
<td>3,237</td>
<td>58.664</td>
</tr>
<tr>
<td>500</td>
<td>1,734</td>
<td>578</td>
<td>58.447</td>
</tr>
<tr>
<td>1,000</td>
<td>774</td>
<td>258</td>
<td>55.397</td>
</tr>
<tr>
<td>5,000</td>
<td>109</td>
<td>36</td>
<td>43.694</td>
</tr>
<tr>
<td>10,000</td>
<td>33</td>
<td>11</td>
<td>32.01</td>
</tr>
<tr>
<td>50,000</td>
<td>2</td>
<td>0</td>
<td>4.919</td>
</tr>
</tbody>
</table>

Table 1: Results from Scenario 1 tests using the sequential scheduler.
Table 2: Standard deviation from Table 1.

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>206.959</td>
<td>0.117</td>
</tr>
<tr>
<td>500</td>
<td>12.767</td>
<td>0.045</td>
</tr>
<tr>
<td>1,000</td>
<td>3.5119</td>
<td>0.117</td>
</tr>
<tr>
<td>5,000</td>
<td>0.577</td>
<td>0.269</td>
</tr>
<tr>
<td>10,000</td>
<td>0.577</td>
<td>0.328</td>
</tr>
<tr>
<td>50,000</td>
<td>0</td>
<td>0.094</td>
</tr>
</tbody>
</table>

Figure 10: A graph showing the average times scheduled for game objects in relation to amount of game objects for the scenario 1 sequential scheduler.
4.1.2 Priority scheduler

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2,300</td>
<td>9,968</td>
<td>29,904</td>
<td>58.749</td>
</tr>
<tr>
<td>500</td>
<td>413</td>
<td>1,790</td>
<td>5,372</td>
<td>58.466</td>
</tr>
<tr>
<td>1,000</td>
<td>185</td>
<td>804</td>
<td>2,413</td>
<td>55.596</td>
</tr>
<tr>
<td>5,000</td>
<td>26</td>
<td>115</td>
<td>346</td>
<td>45.131</td>
</tr>
<tr>
<td>10,000</td>
<td>8</td>
<td>37</td>
<td>113</td>
<td>33.198</td>
</tr>
<tr>
<td>50,000</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>5.09</td>
</tr>
</tbody>
</table>

Table 3: Results from Scenario 1 tests using the priority scheduler.

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. aes. sch.</th>
<th>Avg. col. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>48.497</td>
<td>630.771</td>
<td>0.043</td>
</tr>
<tr>
<td>500</td>
<td>6</td>
<td>79.019</td>
<td>0.062</td>
</tr>
<tr>
<td>1,000</td>
<td>2.082</td>
<td>25.968</td>
<td>0.001</td>
</tr>
<tr>
<td>5,000</td>
<td>0</td>
<td>3.215</td>
<td>0.157</td>
</tr>
<tr>
<td>10,000</td>
<td>0</td>
<td>1.528</td>
<td>0.003</td>
</tr>
<tr>
<td>50,000</td>
<td>0</td>
<td>0.577</td>
<td>0.078</td>
</tr>
</tbody>
</table>

Table 4: Standard deviation from Table 3.
Figure 11: A graph showing the average times scheduled for game objects in relation to amount of game objects for the scenario 1 priority scheduler.

4.2 Scenario 2

4.2.1 Sequential scheduler

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Col. GO (%)</th>
<th>Avg. sch.</th>
<th>Avg. col. lpd.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>9,834</td>
<td>3,277</td>
<td>58.705</td>
</tr>
<tr>
<td>484</td>
<td>9.091</td>
<td>1,795</td>
<td>598</td>
<td>58.597</td>
</tr>
<tr>
<td>961</td>
<td>9.677</td>
<td>795</td>
<td>265</td>
<td>56.576</td>
</tr>
<tr>
<td>4,900</td>
<td>10</td>
<td>121</td>
<td>40</td>
<td>47.871</td>
</tr>
<tr>
<td>10,000</td>
<td>10</td>
<td>40</td>
<td>13</td>
<td>39.795</td>
</tr>
<tr>
<td>49,729</td>
<td>9.865</td>
<td>2</td>
<td>0</td>
<td>5.631</td>
</tr>
</tbody>
</table>

Table 5: Results from Scenario 2 tests using the sequential scheduler.
<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>169.63</td>
<td>0.102</td>
</tr>
<tr>
<td>484</td>
<td>40.278</td>
<td>0.122</td>
</tr>
<tr>
<td>961</td>
<td>8.737</td>
<td>0.154</td>
</tr>
<tr>
<td>4,900</td>
<td>2.646</td>
<td>0.063</td>
</tr>
<tr>
<td>10,000</td>
<td>0.577</td>
<td>0.152</td>
</tr>
<tr>
<td>49,729</td>
<td>0</td>
<td>0.041</td>
</tr>
</tbody>
</table>

Table 6: Standard deviation from Table 5.

Figure 12: A graph showing the average times scheduled for game objects in relation to amount of game objects for the scenario 2 sequential scheduler.

4.2.2 Priority scheduler
<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Col. GO (%)</th>
<th>Avg. aes. sch.</th>
<th>Avg. col. lpd.</th>
<th>Avg. col. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>2,343</td>
<td>10,067</td>
<td>30,201</td>
<td>58.690</td>
</tr>
<tr>
<td>484</td>
<td>9.091</td>
<td>437</td>
<td>1,896</td>
<td>5,689</td>
<td>58.664</td>
</tr>
<tr>
<td>961</td>
<td>9.677</td>
<td>192</td>
<td>766</td>
<td>2,499</td>
<td>57.132</td>
</tr>
<tr>
<td>4,900</td>
<td>10</td>
<td>29</td>
<td>127</td>
<td>384</td>
<td>49.997</td>
</tr>
<tr>
<td>10,000</td>
<td>10</td>
<td>10</td>
<td>44</td>
<td>132</td>
<td>41.244</td>
</tr>
<tr>
<td>49,729</td>
<td>9.865</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>5.982</td>
</tr>
</tbody>
</table>

Table 7: Results from Scenario 2 tests using the priority scheduler.

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. aes. sch.</th>
<th>Avg. col. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>72.404</td>
<td>719.925</td>
<td>0.104</td>
</tr>
<tr>
<td>484</td>
<td>4.041</td>
<td>56.190</td>
<td>0.117</td>
</tr>
<tr>
<td>961</td>
<td>3.464</td>
<td>47.384</td>
<td>0.306</td>
</tr>
<tr>
<td>4,900</td>
<td>0</td>
<td>4.583</td>
<td>0.205</td>
</tr>
<tr>
<td>10,000</td>
<td>0</td>
<td>0.577</td>
<td>0.135</td>
</tr>
<tr>
<td>49,729</td>
<td>0</td>
<td>0.577</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Table 8: Standard deviation from Table 7.
Figure 13: A graph showing the average times scheduled for game objects in relation to amount of game objects for the scenario 2 priority scheduler.

### 4.2.3 Angle scheduler

<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Col. GO (%)</th>
<th>Avg. aes. sch.</th>
<th>Avg. col. lpd.</th>
<th>Avg. col. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>6,240</td>
<td>2,130</td>
<td>6,392</td>
<td>58.778</td>
</tr>
<tr>
<td>484</td>
<td>9.091</td>
<td>1,155</td>
<td>400</td>
<td>1,200</td>
<td>58.662</td>
</tr>
<tr>
<td>961</td>
<td>9.677</td>
<td>523</td>
<td>180</td>
<td>541</td>
<td>57.731</td>
</tr>
<tr>
<td>4,900</td>
<td>10</td>
<td>81</td>
<td>27</td>
<td>83</td>
<td>48.592</td>
</tr>
<tr>
<td>10,000</td>
<td>10</td>
<td>28</td>
<td>9</td>
<td>28</td>
<td>39.654</td>
</tr>
<tr>
<td>49,729</td>
<td>9.865</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5.64</td>
</tr>
</tbody>
</table>

Table 9: Results from Scenario 2 tests using the angle scheduler.
<table>
<thead>
<tr>
<th>No. of GO</th>
<th>Avg. aes. sch.</th>
<th>Avg. col. sch.</th>
<th>Avg. FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>75.624</td>
<td>73.323</td>
<td>0.036</td>
</tr>
<tr>
<td>484</td>
<td>50.895</td>
<td>56.630</td>
<td>0.061</td>
</tr>
<tr>
<td>961</td>
<td>9.539</td>
<td>9.165</td>
<td>0.117</td>
</tr>
<tr>
<td>4,900</td>
<td>0.577</td>
<td>1.155</td>
<td>0.169</td>
</tr>
<tr>
<td>10,000</td>
<td>0.577</td>
<td>1.155</td>
<td>0.091</td>
</tr>
<tr>
<td>49,729</td>
<td>0</td>
<td>0</td>
<td>0.062</td>
</tr>
</tbody>
</table>

Table 10: Standard deviation from Table 9.

Figure 14: A graph showing the average times scheduled for game objects in relation to amount of game objects for the scenario 2 angle scheduler.
Figure 15: A graph showing the difference in average times scheduled between aesthetic and collider objects in relation to amount of game objects for the scenario 2 angle scheduler.
5 Discussion

Based on section 1 and the results presented above, the following discussion section emerged, to compare scheduler results, possible errors, and discuss the feasibility of using these types of solutions. Future work is also discussed.

5.1 Scheduler results

When using the sequential scheduler, as seen in Table 1 and Figure 10 as well as in Table 5 and Figure 12, all objects are scheduled equally, by design. In these two cases, all game objects were scheduled an average of \(9,773 \pm 61\) (between the two scenarios, when there were 100 game objects). Table 2 and Table 6 show that the difference factor between the two cases \((61 * 2)\) is less than the standard deviation between tests in both scenarios. This means that for 100 game objects scenarios 1 and 2 can be considered to have about equal results regarding the amount of times game objects were scheduled. The trend follows as the number of game objects increases, and the amount of collider objects in scenario 2 is close enough to 10% for this to be a reasonable approximation. This is most likely due to the fact that objects are updated regardless of whether or not they are not in the viewing frustum. However, scenario 2 does show an increased average FPS between scenarios. This increased FPS is most likely due to the fact that objects that are not within the view frustum do not need to be rendered. Rendering calls can be time costly, and in scenario 2 fewer objects are being rendered simultaneously.

For the priority scheduler, we can see a clear rift between the average amount of times collider objects (high priority objects) and aesthetic objects (low priority objects) were scheduled in Table 3 and Figure 11 as well as Table 7 and Figure 13. The text in the implementation methodology states that in these scenarios with the priority schedulers, high priority objects had priority 0 and low priority objects had priority 12. This means that the aesthetic objects should be scheduled 13 times less than the collider objects, which is correct and can be seen in the corresponding tables, Table 3 and Table 7. In the same way as done with the sequential scheduler, the results between the two priority schedulers of scenario 2 are about the same, with the exception of the FPS. The FPS is slightly higher in Scenario 2, most likely for the same reasons as the sequential scheduler has higher FPS in scenario 2, less objects rendered on screen simultaneously.
As seen in Table 9 and in Figure 15 that for each test the average times a collider object was updated is slightly higher than the average times an aesthetic object was scheduled. This might be due to the fact that as the simulation starts it has something that could be described as a “warm up period” before it starts working at full efficiency. The warm up period is a time during the start of the simulation where the simulation is not running at full performance. Why this occurs is probably due to that the operating system allocates processing power dynamically, that is to say the operating system notices the simulation needs more processing power after a few seconds. This warm up period is why the FPS recording starts 5 seconds into the simulation. Why this warm up affects the angle scheduler and not the others, is because as the test starts the camera object is faced away from the collider objects. When the warm up period has ended, the camera object has already rotated closer to the collider objects. This means that the collider objects will be scheduled when the test is running at full capacity.

5.2 Comparison between schedulers

5.2.1 Sequential vs. Priority

When the sequential and priority schedulers are compared we see in Table 1 and Table 3 that if the amount of both objects are multiplied with their corresponding average times scheduled, the result will be the total amount of scheduled during the tests. For the test with 100 game objects, we see that the sequential scheduler scheduled a total of $9,712 \times 100 = 971,200$ times, while the priority scheduler scheduled a total of $2,300 \times 90 + 29,904 \times 10 = 506,040$, substantially lower than the sequential scheduler. This difference is because of how the priority scheduler was implemented. Regardless of whether or not a game objects is to be scheduled, the scheduler iterates through it. This rift could be improved with an alternative method for scheduling, so that the scheduler does not need to iterate through game objects that are not to be scheduled.

There is a small difference in FPS between the sequential scheduler and the priority scheduler in both scenarios. This is possibly because the priority scheduler checks if it is time to give the process back to the engine whether an object was scheduled or not. Since checking whether an object should be scheduled is a unitary operation, the scheduler will check if the current time exceeds the limit more often, giving it a higher chance of giving the process
back to the game engine sooner than the sequential scheduler.

### 5.2.2 Sequential vs. Angle

Similar to the calculations done for the comparison of the sequential and priority schedulers, we calculate the total amount of objects scheduled in scenario 2 for the sequential and angle schedulers. The sequential scheduler scheduled \(9,834 \times 100 = 983,400\) objects in the first test, while the angle scheduler scheduled \(6,240 \times 90 + 6392 \times 10 = 625,520\) game objects. This difference is motivated in the same way as the difference between sequential and priority schedulers, and this difference stays throughout the different tests with increasing amount of game objects. It is also possible that this difference is due to the complexity of the scheduling algorithm, as the angle scheduler must check whether a game object is in front of the camera object or not when deciding its priority; making it the most complex and time consuming of all the three schedulers.

There is a very slight difference in average FPS between the angle scheduler and the sequential scheduler, as seen in Table 5 and Table 9. The angle scheduler has a higher FPS, with the same motivation as the difference between sequential and priority, that the angle scheduler will check more often whether it is time to give the process back to the engine, or not.

### 5.2.3 Priority vs. Angle

As done before, the total amount of objects scheduled will be compared, however this time between the priority scheduler and the angle scheduler. We know that in the first test the angle scheduler scheduled 625,520 objects, and in Table 7 we see that the priority scheduler scheduled \(2,343 \times 90 + 30,201 \times 10 = 512,880\) objects in the first test of scenario 2. This difference is most likely due to the fact that the priority scheduler’s low priority is 12 while the angle scheduler’s priority is 7. By doing a few calculations we can calculate how many objects would have been scheduled if the priority scheduler had a low priority of 7, specifically \(\frac{12}{7} \times 2,343 \times 90 + 30,201 \approx 663,501\) objects. This means that if the priority scheduler and angle scheduler used the same low priority then the priority scheduler would schedule more objects than the angle scheduler. This is most likely due to the fact that the angle scheduler is more complex and spends more time scheduling than the priority scheduler.
By studying Table 7 and Table 9, we can see that there are small differences in FPS. For tests up to 1,000 objects the angle scheduler has a slightly higher FPS, however for tests with more game objects then the priority scheduler has a higher FPS. Looking at the standard deviations of these tests in Table 8 and Table 10 we can conclude that the angle scheduler only has a higher FPS in cases where the results difference between the results are lower than the standard deviation, making this higher FPS in the first tests probably coincidental. In the later tests the priority scheduler has a higher FPS due to it having a simpler scheduling algorithm, allowing for more checks to see if the process should be given back to the engine. Why this difference does not show in the earlier tests can be explained by the fact that when there are less game objects in the simulation it is less taxing on the FPS, making the results of FPS stay around Godot’s max FPS: 60.

5.3 Accuracy of results

There are factors that could be used to argue that the results and conclusions are not general, and only specific to this situation. Firstly, these tests were only conducted on one machine, which means one cannot be sure that the results would be the same on another. Secondly the tests required a warm up period of a few seconds, and the results (except for FPS) were recorded partly during this warm up period. This could be used as an argument that the results are unrepresentative of an actual simulation, as the warm up period is minimal and does not affect a majority of the simulation. Finally, this study did not test the angle scheduler in scenario 1, where all game objects are on screen at once. While this may not seem relevant for a camera based scheduling system, there are situations where mobile cameras become static, and should therefore also have been tested.

However, these results can easily be reproduced and the tables for standard deviation of the different test results showed a small variance in comparison to the corresponding results, meaning that the warm up period was close to constant between tests. This means that these automated tests can be repeated with a relatively high level of predictability. While these test results may vary slightly depending on the system specifications, they have shown a dependability that defends the results as general instead of specific to this situation. Scenario 1 is a simplification of a typical game setting, where there is a static camera showing the entirety of the simulation (often done in strategy or puzzle games), and the conclusions drawn from the results could
be applicable to these types of games. Scenario 2 showed that results did not vary much when using a camera object among the game objects. This means that a priority based scheduling system could be applicable regardless of where the camera object is placed.

5.4 Feasibility

While the priority scheduler has shown an increase in performance for prioritized game objects, there may be some reasons as to why one would not want to implement such a scheduler in a game. It is possible that using a priority based scheduler could increase the performance of high priority objects to unnecessarily high levels, while low priority objects receive a loss in performance that affects the overall user experience. With a scheduler like this, the time step between schedules is not as predictable as in a sequential loop, making it vital for the programmer to implement correction (position or otherwise) where it is needed. This means that there would be a bigger burden on the programmer when using a system like this. When using a camera based priority scheduler the game may execute unpredictable behavior as the player could move the camera in ways that creates an ineffective prioritization.

The feasibility of using a priority based scheduler could be discussed based on the results of this study. As mentioned earlier, the priority scheduler did give an improvement to prioritized game objects, about 3 times as many schedules as in the sequential solution. It also gave a slight FPS increase. However, the total amount of game objects scheduled was lower than in the sequential solution. The increased complexity of using such a system is not unreasonable as many situations in game development already need to use the time step for correction. Using this type of solution might cause unpredictability, since the scheduler could have to wait until it receives the process from the game engine before scheduling the next object. Therefore, if a priority-based scheduling algorithm is used, one should possibly make sure that objects that are vital to gameplay are updated independently of the scheduler, to make the game more reliable. With all the information discussed above, the priority scheduler could be feasibly used in certain game situations, particularly ones where certain objects are more important and predictability is not of high priority.

Using the angle scheduler showed that using one may lower the average amount of game objects scheduled compared to the sequential baseline. The
angle scheduler performed equal to the priority scheduler when it came to FPS in situations where there were up to 1,000 game objects. While player experience was not something measured in this study, when testing the angle scheduler there was noticeably less lag in during the demanding tests. However the angle scheduler did not make a notable improvement in performance, as the scheduling algorithm was the most complex of the three implemented schedulers. The angle scheduler may be feasible to use in game situations where there are fewer objects on screen, with a very mobile camera, but may be unsuited for use in games where the camera is static or shows a majority of the game objects in the game.

5.5 Future work

Future work could include more game scenarios to test the scheduling systems in a wider variety of situations that could occur in commercial games. One could also attempt to implement schedulers in another engine, or construct an engine based on priority scheduling. A clear improvement to the existing implemented schedulers would be to try to decrease checks for objects that will not be scheduled, and use a queuing system instead. One could also try to create other schedulers that base priority on how much of the screen area is being taken up by a certain game object, or how close a player object is to another.
6 Conclusion

Based on the discussion in the previous section, regarding the following research questions mentioned in section 1, conclusions can be drawn.

- Does adding priority-based scheduling for game objects in the commercial Game Engine Godot increase, or decrease, performance of prioritized game objects and frames per second?

When using the priority scheduler there was an increase in performance of prioritized game objects and FPS in all of the tests. However, the angle scheduler did not greatly increase performance or FPS.

- Does using a priority-based scheduler affect total performance of both low priority and high priority game objects?

Both the priority scheduler and the angle scheduler decreased the total amount of objects scheduled on average between the tests. In this implementation both the schedulers affected this type of performance negatively. Improvements were discussed in the Discussion section, that could potentially increase the priority based schedulers’ overall performance.
Appendices

A Glossary

- Computer/Video Game - An interactive piece of entertainment that is played primarily using computational technology.

- Sprite - A two dimensional graphic, usually representing an object with interaction. Can be animated as well.

- Mesh - A three dimensional graphic model. These can be animated and can be likened to a 3D sprite.

- Rigidbody - A component in many game engines that allow for an object to have physics behaviors, such as be affected by gravity.

- Game Object (alternative titles) - An object within a game, usually representing some kind of object, with a single mesh or sprite. Alternative titles include: Pawn/Actor (Unreal Engine), Scene/Node (Godot), Object (Game Maker Studio).

- Scene - Usually denoting a virtual area, 2D or 3D where the game can be simulated. In Godot a scene can also refer to a game object.

- Polygons and triangles (specific to meshes) - Meshes are usually built out of polygons put together to form a 3D shape. These polygons are broken down internally into triangles (sometimes called tris) for easier handling by the system.

- AI - Artificial intelligence. Defines the behavior of the game objects, usually a non-player character within a game.

- Gameplay - The interactive portion of the game.

- View/Viewing frustum - The part of the game being rendered onto the screen.

- Camera object - This object only exists in a few game engines, but is the object that defines the view frustum.

- Frames per second (FPS) - The amount of times a game or simulation is rendered to the screen every second.
B  Highlighted Pseudocode

Listing 3: Pseudocode describing the scheduler

```python
while current_time < start_time + ms_per_update:
    if object.should_schedule():
        object.schedule()
    object = next_object()
    current_time = OS.get_ticks_msec()
```

Listing 4: Pseudocode describing the schedule checks for each scheduler

```python
should_schedule_sequential():
    return true

should_schedule_priority():
    if times_skipped == priority:
        times_skipped = 0
        return true
    times_skipped += 1
    return false

should_schedule_angle():
    priority = high_priority
    if camera.is_position_behind(object):
        priority = low_priority

    if times_skipped >= priority:
        times_skipped = 0
        return true
    times_skipped += 1
    return false
```
References


DeNero, J. (2014), Composing Programs, UC Berkeley.


Gregory, J. (2009), Game Engine Architecture, Taylor and Francis Group, LLC.


Nystrom, R. (2014), Game programming patterns, Genever Benning.


