DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG

Projeto Orientado em Computação I (DCC604)

Proposta Mista

Analysis of Game Performance at Different Abstraction Levels

Kael Soares Augusto

Advisor: Luiz Chaimowicz

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte, MG, Brazil
April 5, 2024

1. Introduction

Video games are complicated programs that pose many challenges specifically to programmers, including:

- Performance - Thanks to game's constantly looping structure and their reliance on all computer components
- Structure - Game code structure is complex, with many objects, phases and pipelines that often break, slow down or take time to develop
- Size - Games tend to be big, thanks to the requirement of representing up to thousands of characters, behaviors, skins or environment rules

Over the years, in an attempt to reduce the barrier of entry into game programming, many game engines have entered in differing levels of abstraction to help new coders get into game development. Figure 1 contains some of the most popular Game Engines and libraries in the market.

Figure 1: Compendium of popular game engines and libraries

However, there are very few studies on how much these levels of abstraction help or complicate the development of games. In general programs, abstractions tend to alleviate the challenges of structure and size, reducing development time and complexity. Nonetheless, any level of abstraction may come at a cost of performance.

The objective of this project is to compare, at three different levels of abstraction, the performance of similar game prototypes, while documenting the ease of coding in quantitative measures such as lines of code and qualitative measures such as complexity of the structures used.

This project will be divided in two parts. This document refers to the first part of the project, which, as described above, will compare performance in general game prototypes between levels of abstraction. The second part of the project will be responsible for going more in-depth into comparing one or more algorithms commonly used in games, like path-finding, chunk meshing and more, in differing levels of abstraction.

2. Background

There are two important background concepts for the understanding of this project's significance: **What are Game Engines & their purpose** and **what is abstracted away with Game Engines.** These two concepts will be tackled in Section 2.1 and Section 2.2 and are geared for the first and second parts of the project respectively.

2.1. Game Engine

A game engine is a software framework primarily designed for the development of video games and generally includes relevant libraries and support programs such as a level editor [Val+16]. These engines vary a lot in their level of abstraction, from adding libraries that allow the user to print to the screen up to having their own physics systems, frame creation and programming languages.

Engines that include a Graphical User Interface, such as Unity, Unreal Engine and Godot, are amongst the top in usage. All of them work using a structure similar to the Object Oriented coding paradigm, where game components, such as levels, characters and more are objects or nodes declared by the developer. These are representative of the highest level of abstraction.

Between the engines that don't have a Graphical User Interface, there are many who still provide the user with their own structure for objects in the game. These structures can vary depending on the engine. As an example, Bevy is a game engine based on the Entity Component System. It still provides the user with 2D and 3D renderers, scenes, and more. These are representative of a medium level of abstraction.

Finally, there are game development libraries that expect the developer to make their own structures, providing the user only with functions that allow drawing into the screen, outputting sounds and possibly math libraries for physics and 3D rendering. These often call themselves libraries, because they are closer to a thin wrapper over system calls than an engine. However, for the sake of simplicity, they will also be referred to as game engines from now on. Examples include Raylib, Allegro and SDL.

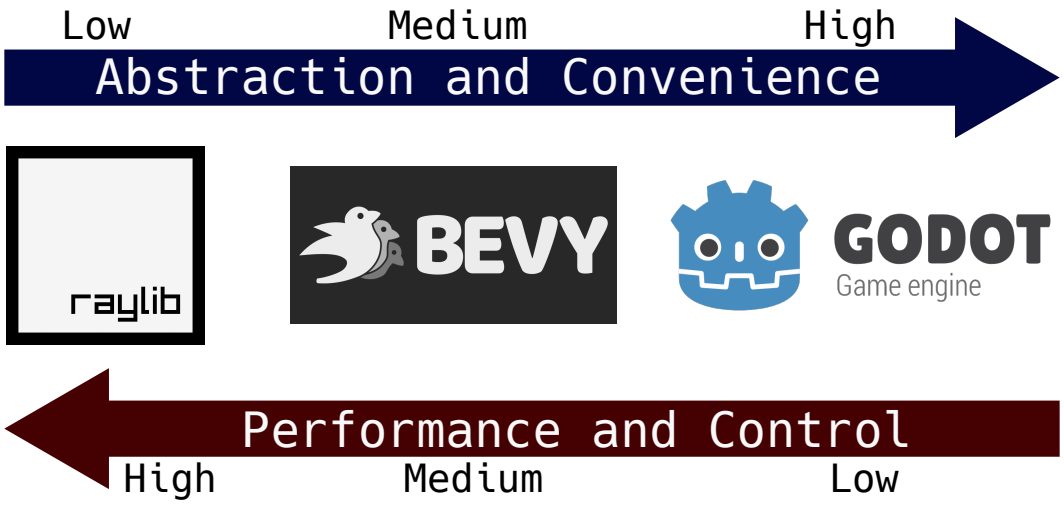

Figure 2: Abstraction and Convenience vs Performance and Control

Developers may choose to forego the conveniences that come from layers of abstractions in favor of more control of their code and better performance thanks to developing in a way that is more geared to their specific use case. This idea is summarized in Figure 2. This trade-off is not absolute, as it can change depending on game scope, game genre, team size and expertise. However, it tends to be a valid general rule for most cases.

In conclusion, the purpose of game engines is to provide layers of abstraction that make development quicker, cheaper and manageable while still attempting to maintain a reasonable level of performance.

2.2. Abstracting Complexity

In his article, [LJ02] demonstrates the importance of game engines in graphical computing by saying that the most complicated pipelines for rendering and the most responsive and interactive simulations can be found not in the most expensive equipment, but instead in consumer computers thanks in great part to game engines.

This is argued because ”The cost of developing ever more realistic simulations has grown so huge that even game developers can no longer rely on recouping their entire investment from a single game.” [LJ02]. The cost mentioned is not only in money, but also time. Recreating complicated pipelines, rendering techniques, physics engines and much more is both complex and costly.

In Section 2.1 it was mentioned that the purpose of game engines is to reduce the barrier of entry into game programming. This is done by abstraction of the previously mentioned complicated pipelines into reusable snippets, libraries, functions and interfaces. Even the most simple game engines, such as Raylib, still have abstractions that hide away system calls, output creation, GPU code and more.

However, using a complicated game engine is not always desirable. As an example, consider Minecraft [Per09], a game made in Java using LWJGL (low abstraction). For it’s time, Minecraft was a computationally intensive game that used a lot of resources.

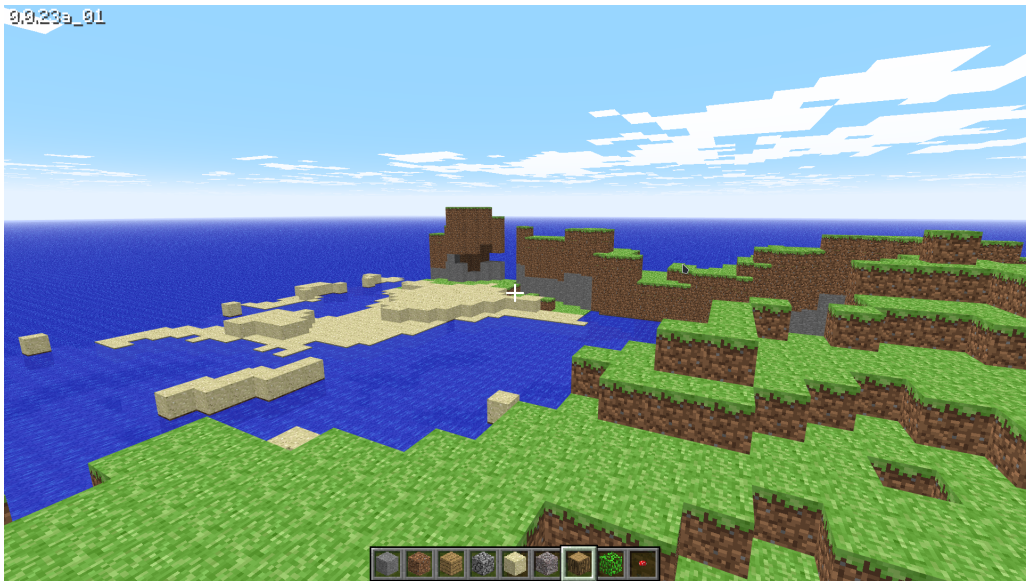

Figure 3: Image of an old version of Minecraft

One of the main techniques used to make Minecraft viable is **Chunk Meshing**. Notice in Figure 3 that the game is made of cubes, called voxels. If we were to attempt to render every possible voxel's 6 faces, including those underground, the game would slow down significantly. For that reason, Minecraft creates a Mesh of voxels, rendering only the visible exterior of this mesh. A more detailed explanation is given by [mik12].

When considering the complexity of this algorithm, adding in the aspect of invisible or transparent blocks, moving structures and more, we end up with a hard to implement solution that is drastically important for performance. When abstracting away details, generalizations such as rendering every object requested completely may happen at the cost of performance.

Nowadays, there are game engines that have techniques such as chunk meshing already included into their engine. However, even more performance can be gained by assuming that, for example, your world is mostly stable, not requiring updates in the mesh generated so often. For every possible abstraction there may be a way to specify its implementation for a specific use case that may be faster.

In conclusion, there are complex algorithms that can gain a lot in performance, but also be costly to learn, develop and maintain. These algorithms can always be generalized and abstracted, but there is a loss of control over its details and there is a limit to how many different techniques are abstracted before an engine becomes bloated with features, requiring some games to create their own engines from scratch to achieve ideal performance.

3. Methodology

This section describes exclusively the methodology for the first part of the project. It is more representative of what this project will do and why it should be done in this manner. For a more detailed breakdown of tasks done during the project, see Section 5.

3.1. Levels of Abstraction

Given time constrains, there can't be a comparison between all possible levels of abstractions, nor can there be a valid measurement of how much more abstract an engine is than another. Given these two facts, this project will study 3 levels of abstraction with clear differences between themselves:

- Low Abstraction - Represented by only having simple functions for input of keyboard and output of images, sound and metrics.
- Medium Abstraction - Represented by having its own structured system and engine that manages interactions between components.
- High Abstraction - Represented by turning all components into objects with a fully featured engine that can make games while having its own UI and/or language.

To represent each level of Abstraction, one open-source game engine shall be chosen from the ones listed in Section 2.1. The final choices for the three representatives are:

- Raylib - Low Abstraction - Using native C/C++ structures
- Bevy - Medium Abstraction - Using Entity Component System structures
- Godot - High Abstraction - Using Godot's Node structure

These game engines have already been shown in Figure 2 representing their levels of abstraction.

3.2. Measurements

Comparing programs and benchmarking across different languages and engines is no easy feat. This problem is two-fold:

1. We must avoid biases inserted by different programmers, games and styles.
2. We need proper tooling and universal metrics to gather data and compare between engines.

1. Coding From Scratch: To avoid the biases that may arise from comparing different games made by different people in differing styles, all of the games made in this project will be coded from scratch or hand-picked from open-source examples in an attempt to represent each level of abstractions, their tools and how they resolve problems in the most faithful way.

2. Tooling: To compare results between different engines, there is also the need to decide on metrics that are representative and useful (More on this in Section 4) and the need to create the tooling necessary for those measurements. This too is a part of the project and has its own time allocated in Section 5.

4. Expected Results

4.1. Performance

Normal programs tend to be difficult to benchmark and compare, as there are few universal metrics, and these metrics may vary depending on what the user wants. For example, a program may run faster while being less precise than a competitor that is slower but more precise. However, games do have a universal metric of performance that is desired and can be used: Frames per Second.

Although there is no tool that universally measures Frames per Second, all game engines that were chosen have the possibility of gathering the amount of time between frames. With proper logging and parsing this can be transformed into values of frames per second of every second across the execution time of the program.

It is expected that as we use higher and higher levels of abstraction, we end up loosing more and more performance. However, not necessarily this loss is noticeable for the end consumers, as minimal differences aren't problematic. Discovering just how much can be gained by removing layers of abstraction is the objective of these measures.

4.2. Ease of Coding

Although the title of this project is "Analysis of Game Performance at Different Abstraction Levels", it is important to contrast the analysis of performance between abstraction levels with the advantages that abstraction brings to programmers. One of the main advantages of abstraction is turning easier the creation of code. This is represented by "Convenience" in Figure 2.

Ease of Coding is mostly a subjective measure, as it can change depending on more than just the language or engine, such as: the developer's mood, ambient, working environment, project and more. However, there are fair ways to compare and discuss ease of coding.

One way to do this boils down to a detailed description of subjective measures, such as: structures used, complexity of the language or engine, difficulty of envisioning/designing the final project and more. Another way to do this only dealing with quantitative measures, such as development time and lines of code.

During the development of this project both of these strategies will be explored for a more faithful representation of the subjective and objective measurements of the experience of using a higher level of abstraction.

It is expected that, as we use higher and higher levels of abstraction, we end up with an easier environment to develop games. However, more abstraction is not necessarily good, and it can lead to an alienation of the inner workings of your game. Discovering just how much can be gained by adding layers of abstraction is the objective of these measures.

5. Timeline

This timeline represents expected dates and developments across the next weeks. However, the timeline is subject to change as new discoveries are made to better align the time spent with the expected results. The first week starts with 07/04/2024 and the last week ends at the final due date at 27/06/2024.

Figure 4: Calendar with Important Due Dates

Every due date is mentioned before the week that is responsible for producing its deliverable. The tasks related to this document, such as the choosing of the project theme, writing this document and planning of the project have already been done, and thus they shall not be included in the timeline.

Current Date: Week 0 05/04 - Delivery of this Document

- Week 1 - Set-up Raylib, Bevy and Godot environments. Document this process.
- Week 2 - Set-up a organized git repository with all three game engines and possible tools used.
- Week 2 & 3 - Create basic game prototypes, such as drawing basic forms, basic movement and collision on all three game engines.
- Week 4 - Create tools to compare games in lines of code and frames per second in all three engines.

First Due Date: Week 6 13/05 - Partial Pitch

- Week 5 - Compare all prototypes created. Summarise, write down and report findings in first pitch video.
- Week 6 to 8 - Implement one full or many more small game prototypes in all three engines.
- Week 10 - Measure, compare and report results in a summarised manner.

Second Due Date: Week 11 21/06 - Final Pitch

- Week 11 - Construct final project pitch, its findings and conclusions.

Third Due Date: Week 12 27/06 - Final Report

- Week 12 - Write and deliver the Final Report of this project.

References

[LJ02] Michael Lewis and Jeffrey Jacobson. “Game engines”. In: *Communications of the ACM* 45.1 (2002), p. 27.

[Per09] Markus Alexej Persson. *Minecraft*. 2009.

[mik12] mikolalysenko. *Meshing in a Minecraft Game*. 2012. URL: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. Accessed 2024-04-05.

[Val+16] Rafael Valencia-García et al. *Technologies and Innovation: Second International Conference, CITI 2016, Guayaquil, Ecuador, November 23-25, 2016, Proceedings*. Springer, 2016.

[Bev] Bevy. *Bevy Engine*. URL: https://bevyengine.org/. Accessed 2024-04-04.

[God] Godot. *Godot Engine - Free and open source 2D and 3D game engine*. URL: https://godotengine.org/. Accessed 2024-04-04.

[Ray] Raylib. *raylib — A simple and easy-to-use library to enjoy videogames programming*. URL: https://www.raylib.com/. Accessed 2024-04-04.