Skullamanjaro Report
Michael Wilson
14-5-2020

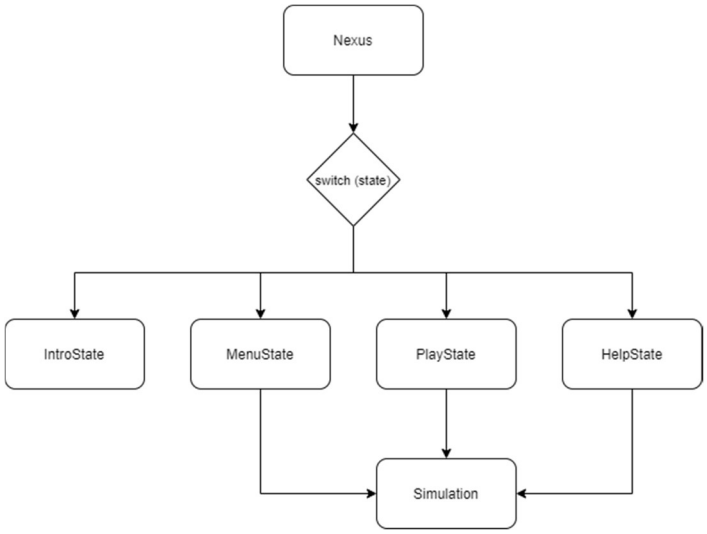
Introduction

Skullamanjaro is a video game developed for the PlayStation Vita and Windows PC. It was written using C++, the GEF framework, and the Box2D physics library. The objective of the game is simple, the player bounces skulls on a pillow to score points before time runs out. Users can move their pillow left and right using the arrow keys. Skulls appear above the screen which then fall under the force of gravity. If the player can position their pillow such that a skull lands on it, the skull will bounce upwards and the player will be rewarded with a point. If a skull is allowed to fall past the bottom of the screen, the player will be penalized five points. The player only has fifty seconds to score as many points as they can. Often the score will be negative. Once the timer runs out, the game ends.

When the game is initially launched, the user is greeted with a splash screen, followed by the main menu. The main menu has five options: “Start”, “How to Play”, “Difficulty”, “Volume”, and “Quit.” The “Start” option does exactly what one might expect, it starts the main game using the currently set difficulty. “How to Play” is an easier version of the main game with only one skull, no timer, no music, and instructions displayed on the screen. “Difficulty” allows the user to change the difficulty of the game with the left and right arrow keys. There are three difficulty levels: “PEON” (Easy), “HERO” (Normal), and “LEGEND” (Hard.) “Volume” allows users to adjust the game’s audio volume between 0% and 100%. And finally “Quit” closes the game entirely.

Application Design

The program is structured around a central class called *Nexus* which handles initializing, rendering, updating, and disposing the current game state. The game consists of four main states: *INTRO* which handles the introductory splash screen, *MENU* which is the main menu, *PLAY* which is the core game, and *HELP* which is the "How to Play" screen mentioned previously. Each state has its own class for better organization.



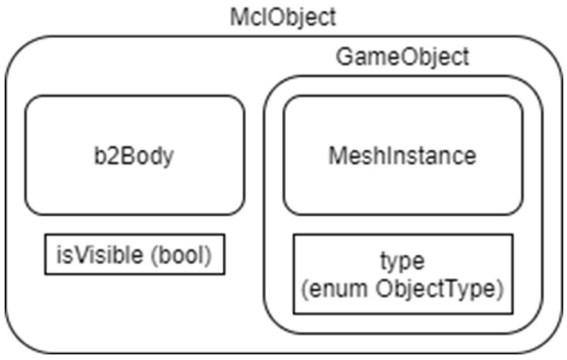
Here is some simplified pseudocode of the *UPDATE(…)* function inside *Nexus*:

```
Nexus::UPDATE()
{
	switch (current_game_state)
	{
		case INTRO: IntrState::UPDATE(); break;
		case  MENU: MenuState::UPDATE(); break;
		case  PLAY: PlayState::UPDATE(); break;
		case  HELP: HelpState::UPDATE(); break;
	}
}
```

The *Nexus* class also contains a number of useful static functions for use throughout all the game states. These include functions like: *GET_RANDOM(…)* for getting a random integer between a minimum and maximum value, *LOAD_SPRITE(…)* for loading a sprite asset from an image file, *DRAW_TEXT(…)* for easily drawing text to the screen, and others.

This program additionally uses a class called *Simulation* to initialize, update, render, and unload the Box2D physics simulation. This class contains a dynamic list of *MclObject*s which represent all entities existing within the Box2D world. *Simulation* also provides some useful static functions like *ADD_ENTITY(…)* for adding a *MclObject* to the simulation, and *GET_ENTITIES()* which returns a pointer to the list of entities currently in the simulation. All game states except *INTRO* utilize the *Simulation* class.

The class *MclObject* contains a *GameObject*, *MeshInstance*, and Box2D *b2Body*. It also provides some functions for construction, updating, rendering, deletion, and modification. Some examples of these functions are: *appyForce(…)* for applying a *b2Vec2* force to the physics body, *setVisible(…)* for setting whether the mesh should be rendered, *setPosition(…)* which moves the object to a new position, and a few others. This object is used throughout the program as an all-encompassing game entity. Here is a diagram of *MclObject*'s basic structure:



As stated previously, each game state has its own class. These are: *IntroState, MenuState, PlayState,* and *HelpState*. Intro state is very simple. It displays the logo of a fictional company "Wizlon Software" on the screen, waits a couple seconds, then calls *Nexus::CHANGE_STATE(MENU)* which takes the user to the main menu.

The *MenuState* class is a bit more complicated. Firstly it contains a struct called *MenuOption* which serves to wrap all the data for a menu option into a single data structure. On *MenuState*'s initialization, it creates an array of *MenuOptions* and populates it with the appropriate data for each of the five options "Start", "How to Play", "Difficulty", "Volume", and "Quit." Lastly, it initializes the *Simulation* class and adds some skulls. This is done to create an interesting background of skulls dynamically falling and colliding with each other.

PlayState houses the core game functionality. In the *INITIALIZE(…)* function, it initializes *Simulation*, creates a number of skulls based on the currently set difficulty. "PEON" has 3 skulls, "HERO" has 6, and "LEGEND" has 12. Then two invisible static *MclObjects* are created on the left and right sides of the screen to ensure skulls cannot fly off that way. And lastly the player entity is created. The player body is actually static but is moved by changing its position when the player uses the left and right arrow keys. The *UPDATE(…)* function handles input and keeps track of how much time has elapsed. It also ensures that when a skull falls below the bottom of the screen, the player is deducted 5 points and the skull is reset to a random position above the screen again.

The *HelpState* is nearly identical to *PlayState* except that no music is played, there is only one skull, the gravity is slightly weaker, there is no timer, and instructions for how to play the game are displayed in the center of the screen.

Techniques Used

This program was designed with an overall design philosophy of modularity. Each section of the game is distinct and separate from all the others. To compliment this technique, static global variables and functions are utilized where communication between game states is necessary. Abstraction was also a priority when starting the project. This was done to avoid dealing with low level framework calls for rendering, loading assets, updating physics, etc. Therefore, classes like Simulation, Nexus, and MclObject were created to handle this low level work. This also minimized redundancy and increased readability.

Using a modular program structure provided some benefits but also created some limitations. The benefits are enhanced readability and simplicity. When all the code involved in initializing, updating, rendering, and unloading each game state is in one single class, the task of finding and modifying said state becomes a great deal simpler for humans. It also makes the job of organizing variables and functions trivial. The obvious major drawback of a modular structure is impaired communication between sections of the game. But this is not much of a problem as much of the data inside say, the HelpState class, is irrelevant to all other game states. There are only a few examples, like changing the difficulty and volume, where communication between states is necessary. When this necessity does arise, it is easily handled with a static global function. There are, of course, functions which are needed by multiple game states. For these shared requirements, the Nexus class provides an assortment of functions for loading assets or changing the current game state.

Another main technique used in Skullamanjaro is abstraction. From the very beginning it was clear that dealing directly with the GEF framework API throughout all the code would create redundancy, reduce readability, and waste time. To handle this problem, classes like Simulation were created to encapsulate most of the necessary functions for managing the Box2D physics simulation. As a result of this decision, the structure of all game states was greatly improved. Additionally, while perhaps inefficient for cache usage (Game Programming Patterns, 2014), the MclObject class encapsulates all the necessary data for managing an entity within the game world. This ensured that when creating game states, it was not necessary to deal with each individual component for each entity. Instead, with one simple function call, an entity will be added to the game with all the necessary components already initialized and attached.

Finally, the file ‘Nexus.h’ reflects another design method used in this project: centralization. The ‘Nexus.h’ file stores headers for the Simulation class, the Nexus class itself, and all the game state classes. This was done so all game states could both access each other when needed and access all the static functions in the Nexus and Simulation classes. Moreover, all ‘include’ and ‘using’ statements used throughout those classes are declared inside the ‘Nexus.h’ file only. This again represents a design philosophy of centralization within this program.

User Guide

Upon launching the application, you will see a short splash screen. After a few moments you will be taken to the main menu. Inside the main menu you have five options: Start, How to Play, Difficulty, Volume, and Quit. Use the up and down arrow keys to navigate these options. Use the enter key to select an option. For the Difficulty and Volume options, you can use the left and right arrow keys to adjust their values. Below are explanations of the five options:

- Start
    - Begin the main game using the currently set difficulty.
- How to Play
    - A practice room with instructions.
- Difficulty
    - Adjust the game difficulty. Use the left and right arrow keys.
    - PEON - Easy
    - HERO - Medium
    - LEGENT – Hard
- Volume
    - Adjust the audio volume. Use the left and right arrow keys.
    - The value ranges between 0% and 100%
- Quit
    - Exit the game.

The objective of the game is to score as many points as possible before the fifty second timer runs out. Points are scored by bouncing skulls off the pillow. Use the left and right arrow keys to move the pillow. Five points will be deducted every time a skull falls below the bottom of the screen. You can return to the main menu at any time by pressing the escape key.

Data Oriented Design

Performance problems typically arise in video games as more features, entities, levels, etc. are added. There are potentially many reasons for this but one of the most common is poor cache usage. Programmers will often design their code with increased levels of abstraction that make it easier humans to understand. However, an unfortunate consequence of this is inefficient CPU usage. Calls to fetch data from memory (RAM) are extremely slow compared to fetching data from the cache. (Game Programming Patterns, 2014) Therefore, it is imperative that programmers design critical aspects of their code to maximize cache usage.

For Skullamanjaro, there are several sections of the code which could be re-written to improve cache usage and boost performance. Luckily, performance issues have yet arisen in the current build. However, if more complexity were to be implemented, it is likely that the current structure would be inadequate. One example is the *MclObject* class which makes life easier when programming, but actually degrades performance. To fix this issue, *MclObject* should be broken down into its subcomponents like *MeshInstance*, *b2Body*, and *GameObject*. These separate components should instead be stored in distinct arrays within the *Simulation* class and looped through individually. This creates a situation where the subcomponents of *MclObject* are stored sequentially in memory and therefore will be cached in large chunks instead of one at a time. It is still possible that each *MclObject* could contain pointers to its respective components, however, it is important that the update and render functions loop through the components directly instead of traversing said pointers.

Some pseudocode which demonstrates this change:

```
// Before
List<MclObject*> entities;
Simulation::UPDATE()
{
	for (int i = 0; i < entities.size; i++) entities[i]->update();
}
```

```
// After
MeshInstance meshList[];
b2Body bodyList[];
GameObject objList[];
Simulation::UPDATE()
{
	for (int i = 0; i < meshList.size; i++) meshList[i].update();
	for (int i = 0; i < bodyList.size; i++) bodyList[i].update();
	for (int i = 0; i < objList.size; i++) objList[i].update();
}
```

Conclusion

This project has taught me a great deal about game design, the asset pipeline, working with existing frameworks, and C++ in general. As someone more experienced in Java, developing a game in C++ proved an interesting challenge. But because of this project, I now possess a much stronger grasp of the underlining C++ programming paradigm. Some specific lessons learned while creating this program are the following:

Do not attempt to approach classes in C++ as you would in Java. This lesson was hard learned but ultimately important for my growth as a developer. With Java, all functions, variables, and subclasses, are stored within the same set of curly braces and within the same file. With C++ on the other hand, classes are much less centralized. This is mostly due to headers. A header is a pre-declaration of a class along with all its functions and variables. This means that a class's functions and variables can be defined anywhere in the code which includes said class's header. This difference, while small, has a big impact when organizing and structuring code.

Creating an asset pipeline is not as straightforward as one might think. It is unwise to approach the problem haphazardly, especially in larger projects. The asset pipeline is essentially how assets like 3D models, bitmap images, fonts, audio, etc., go from raw files to useable objects within a game. The first solution is typically to load the content only when it is needed. However, after some time, it becomes clear that this will not only lead to spaghetti code, but also create unnecessary redundancy. To solve this, it is wise to designate an area in the program where all the content for each section is loaded and made accessible.

To conclude, Skullamanjaro accomplishes the goals it set out to achieve. However, there is always room for more improvement. For example: a variety of objects for the player do deal with would make the game more interesting. These new objects could behave differently and offer different rewards. Another addition might be multiplayer and customization options. The possibilities for improvement are infinite, but time, unfortunately, is not. And that lesson is certainly the greatest to be learned when developing a video game.

References

Game Programming Patterns (2014) *Data Locality.* Available at: http://gameprogrammingpatterns.com/data-locality.html (Accessed: 1 May 2020).