Virtual reality, Graphics and Animation
Project - Task 2

Kristoffer Berg Wilhelmsen

October 10, 2022

Abstract

Sphero BOLT is an advanced, programmable robot with onboard sensors including a compass, gyroscope, accelerometer and a light sensor. By using data from the sensors obtained from the Sphero Edu application, we can create a visual simulation of the robot's movement in a game engine such as Godot. After a development phase using C++ and GDScript, we are left with an imperfect, although functioning simulation. The imperfections present us with an opportunity to refine the simulation in future work.

1 Introduction

This paper concerns project work in the course "Virtual reality, Graphics and Animation", which consists of creating a visual simulation of a Sphero BOLT. A Sphero BOLT is an advanced, programmable robot with onboard sensors. The onboard sensors include a compass, gyroscope, accelerometer and a light sensor [Pedersen and Brustad, 2022].

The main criteria for the task are to record and utilize data from a physical robot, moving on a 2-dimensional plane by using the Godot game engine. The recorded data should include any collisions that may have occurred while recording, and said collisions should be reflected in the simulation. Additional features that have not been explicitly requested by the task description are encouraged in order to add a personal touch.

2 Methods

In broad strokes, the project work has been divided into two main processes; recording data with the physical robot and developing the simulation in the Godot game engine. Additionally, as a minor side-objective, a simple skybox has been created using the modelling software Blender.

2.1 Recording data

The data used in the simulation has been recorded by using the Sphedo Edu windows application with a custom script. The application lets users create and define scripts by using JavaScript and an implementation of Sphero Edu's API [Sphero, 2022]. The script has been run multiple times, yielding different data. This is largely due to the script not implementing any thorough collision detection, resulting in inconsistent behavior. The data that was ultimately chosen to be used in the demonstration differed in the way that it had movement in multiple directions following a collision, as well as multiple collisions. This is as opposed to movement in a straight trajectory where the robot would typically get stuck in a wall, thus not changing its course. The exact script that was used in order to produce the results in this report is attached in appendix A.

2.2 Godot development

In line with the requirements given in task description [Pedersen and Brustad, 2022], the simulation is built upon the Godot game engine, which utilizes C++ as well as Godot's own language GDScript.

C++ has been used to create a custom module in Godot that processes and stores data which has been recorded by the physical robot. First, data is read from six individual .csv files. Afterwards, the data is parsed and stored into separate vectors consisting of structs that match each file's properties. When the data has been stored, it is iterated through to detect and store any collisions. For a time step to be classified as a collision, only two conditions must be true; the y-value of the acceleration must be less than or equal to -1.08, and the total acceleration must be greater than or equal to 1.10. These values have been determined through trial and error, while also consulting fellow student Stian Jakobsen. All the data that is stored in the module can be accessed by using getter-functions in Godot. Each function in the custom module returns either void, bool, float or String. This approach was chosen due to the fact that these are all primitive types that interact well with Godot and was easy to implement.

GDScript's main functionality in the simulation is to animate the scene and trigger events. The entire simulation relies on Godot to create an instance of the custom module, load and iterate through the data. By interpolating with Godot's Tween type, in combination with a timer which simulates each time step, the virtual Sphero is moved around, and its rotational degrees are set corresponding to data from the gyroscope. The camera's position is constantly being updated relative to the robot, resulting in a third-person view. Collisions are inserted into the scene programmatically according to the stored data for visual representation, in addition to playing a sound on impact. Once the simulation is complete, an overlay with instructions on how to restart the simulation and the number of collisions that were detected is made visible.

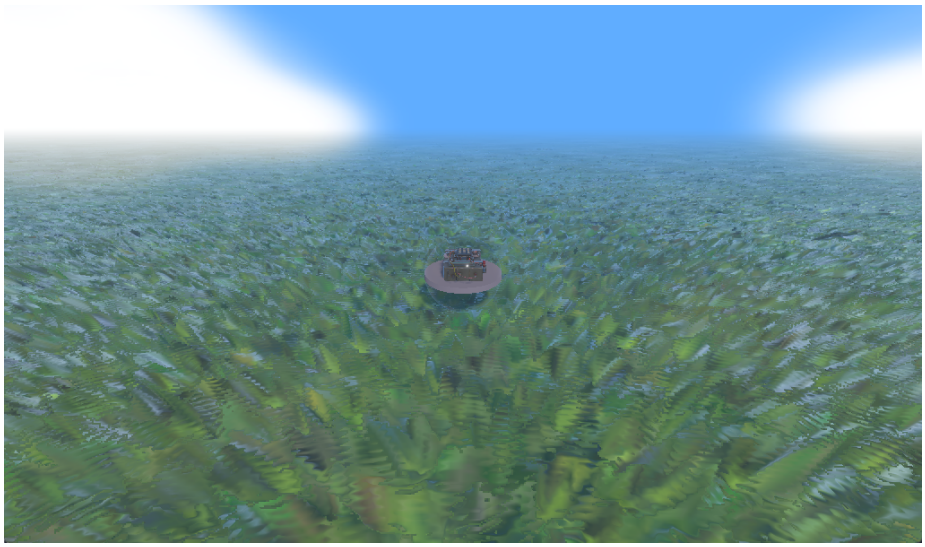

Figure 1: Transparent sphere representing the sphero, a texturized plane and a skybox.

3 Results

Upon instantiation, the main scene as pictured in Fig. 1 is displayed. The scene contains a transparent sphere with a C4 model inside which represents the physical Sphero, a plane texturized with grass that the Sphero will move along, and a skybox.

When the Sphero moves to a location equivalent to where a collision has been detected by the C++ module, a white flag with a brown post is inserted into the scene at the corresponding position as seen in Fig. 2.

As the simulation ends, an overlay as seen in Fig. 3 is made visible. The overlay shows the total number of collisions detected during the simulation, as well as instructions on how to restart the simulation.

A complete summary of the implemented features is as follows:

- Transparent sphere representing the physical Sphero BOLT. The C4 model inside sphere has been provided by Sketchfab [_Vadim2020_, 2022].
- Plane texturized with grass [ambientCG, 2021].
- Simple skybox created in Blender.
- The virtual Sphero moves according to the location-data.
- The orientation of the virtual Sphero is set according to gyroscope-data.
- Superficial collision detection.

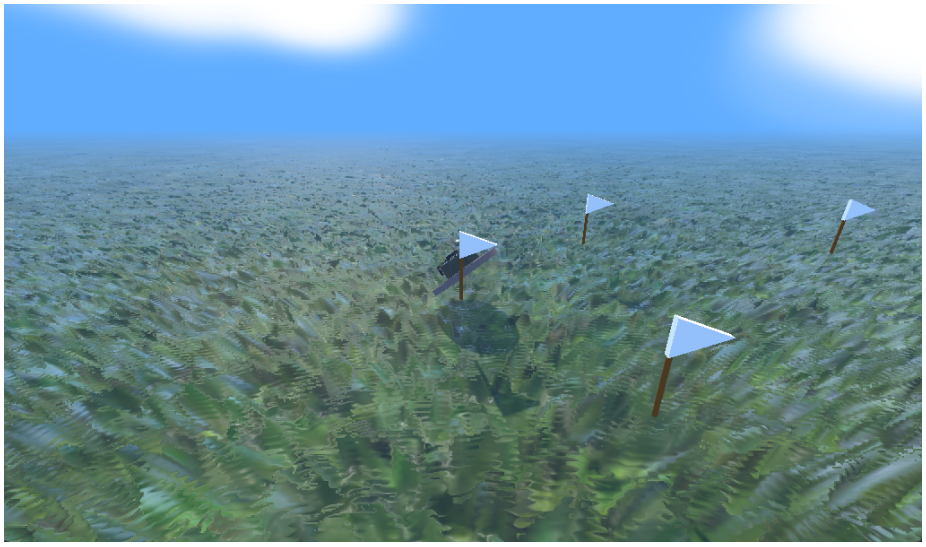

Figure 2: Each of the white flags represent a collision.

Figure 3: Amount of detected collisions and instructions to restart simulation upon completion.

- Flags inserted during runtime, indicating the point of impact for a collision.
- Sound [Sound, 2021] is played on impact of collision.
- Overlay with instructions on how to restart the simulation and the total number of collisions detected upon completion.

4 Discussion

There are a few known, apparent issues with the code. These issues do not deem the simulation useless, however had there been more time, they could have been refined in order to improve the simulation.

The script in appendix A that was used to record data for the demonstration did not implement any thorough collision detection. The Sphero's built-in collision detection proved to have a very high threshold during testing, meaning that a collision would have to occur at full force in order to be detected. Writing a custom script with improved collision detection within the specified timeframe was easier said than done, which made obtaining solid data for simulation purposes a complicated process. Given that the custom script did not implement any further collision detection, the final results were obtained by repeatedly running the script in appendix A until data containing multiple collisions was recorded. This should be taken into account should one try to recreate the data.

Querying data from the custom module could potentially carry a lot of overhead, and should be investigated further. The issue with the current approach is that each function returns a singular value at most (e.g. x-value of velocity), as opposed to returning all the relevant Sphero data simultaneously (e.g. a struct containing data from an entire time step or an array containing all data from every time step). For the current iteration of the simulation, this does not impose an apparent issue. However, should the simulation be scaled up to incorporate a large amount of Sphero's simultaneously, this is a potential bottleneck.

Collisions are detected in the custom module after parsing and storing the Sphero data, however the current collision detection is quite sloppy to say the least. As described in section 2.2, the approach is clearly flawed and severely simplified. However, it somewhat serves its purpose by demonstrating the concept of collision detection, which in turn enables the Godot-side of the simulation to handle collisions. Nonetheless, the C++ module's collision detection accuracy should be improved.

As stated in section 2.2, the change in movement and rotational degrees of the Sphero during the simulation is interpolated using a Tween-node. This is not problematic in itself, however it currently interpolates for each time step found in the data, which is approximately six times per second. This causes certain translations to be abrupt and uneven. A possible solution could be to change the rate of interpolation (e.g. every second or half a second), although this could cause other problems such as the movement being imprecise.

Lastly, a possible improvement that could be made is dynamically rendering the plane's position. The current plane is of size Vector3(1000, 2, 1000). Should the Sphero move outside these dimensions, the plane will not follow, resulting in the Sphero essentially floating. This problem could be solved by rendering a series of planes of smaller dimensions, based on their proximity to the Sphero.

To conclude the paper, we have created a visual simulation of a Sphero BOLT consisting of functionality with varying degrees of completion. While there are imperfections concerning areas such as scalability, collision detection and interpolating movement, the end product is a functioning simulation. The imperfections make up possible areas to focus on for potential future work.

References

[ambientCG, 2021] ambientCG (2021). Grass 001. https://ambientcg.com/view?id=Grass001. Accessed: 21.09.2022.

[Pedersen and Brustad, 2022] Pedersen, A. and Brustad, T. (2022). Task 2: Sphero simulator. https://uit.instructure.com/courses/27060/assignments/78488. Accessed: 03.10.2022.

[Sound, 2021] Sound, E. (2021). Bamboo hit - sound effect (hd). https://www.youtube.com/watch?v=v7-PjPp1JKw. Accessed: 08.10.2022.

[Sphero, 2022] Sphero (2022). Javascript wiki - getting started. https://sites.google.com/sphero.com/sphero-edu-javascript-wiki/getting-started?authuser=0. Accessed: 08.10.2022.

[_Vadim2020_, 2022] _Vadim2020_ (2022). Sketchfab - c4 explosive. https://sketchfab.com/3d-models/c4-explosive-617d754683114b6a8244ececbd03365e. Accessed: 07.10.2022.

Appendix

A Sphero Edu script

```
async function onCollision() {
    stopRoll();
    setMainLed({ r: 255, g: 0, b: 0 }); // set LED to red
}
registerEvent(EventType.onCollision, onCollision);

async function startProgram() {
    for (let i = 0; i < 15; i++) {
        setMainLed({ r: 0, g: 0, b: 255 }); // set LED to blue
        await spin(getRandomInt(45, 180), 0);
        await delay (0.5);
```

```
12              await roll(getRandomInt(0, 360), 255, getRandomInt(1,3)
    );
13          }
14          exitProgram();
15      }
16
```

Listing 1: Custom script for recording Sphero BOLT data.