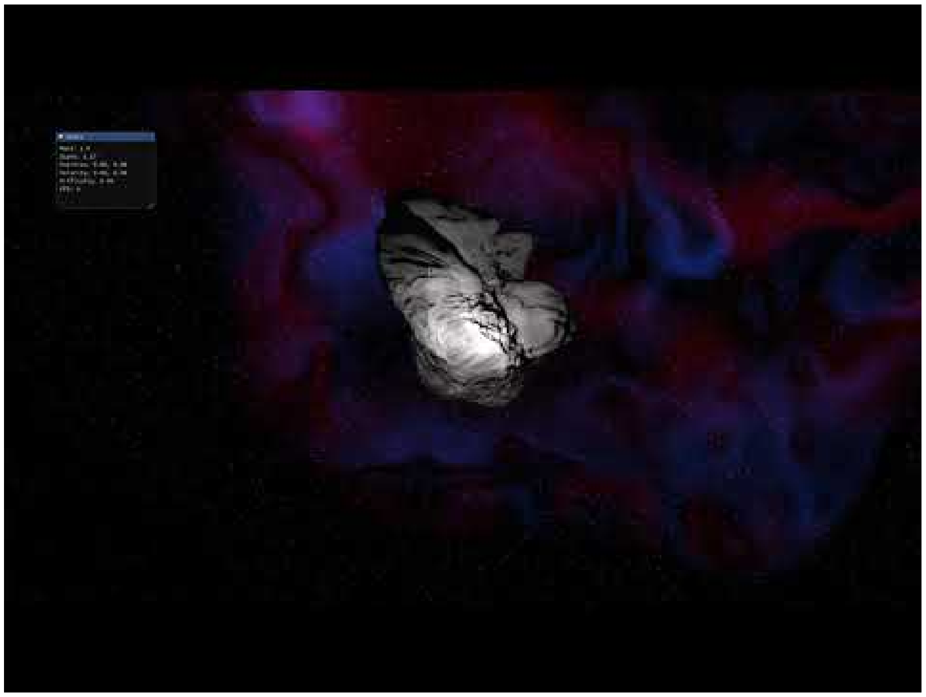
Gravity Smash

Build Development Code Structure Starting Point Inspiration



Gravity smash is a seemingly simple game where the player must smash into smaller objects in order to gain mass, however if they collide with something larger than themselves will die and respawn with half their mass.

The player will start off as an asteroid, floating around with other asteroids, as they gain mass their asteroid will become more spherical, due to the force of it's own gravity.

When an asteroid gets big enough, it will enter a new phase, a dwarf planet. dwarf planets have more colour variation and a smoother surface than asteroids.

In the next stage our dwarf planet graduates into fully grown "planethood", revealing even more colour variation. In a future version I would like to add atmospheres and oceans to these.

The final stage of our planets lifespan is becoming a gas giant. Gas giants have swirling and evolving patterns in their atmosphere. At the end of this period, the planet will reach critical mass and the game will be complete.

Build

Cmake is used as the build tool, as it provides an easy way to include and manage dependencies.

The project requires glfw3, OpenGL, X11 and the threading library tbb.

To build the project on linux, run the following commands: > mkdir build && cd build > cmake ../ > make

After the project is built, you can run the generated executable using:

```
./GameEngine
```

Development

My development ethos has been to start with the most essential features first. To that end I started by rendering a triangle to the screen, then turning that triangle into a spinning cube, and then into a sphere.

Once I had my sphere it was time to add some gravity. I implemented this using a naive n-body simulation, using formula that I learned during my A-Level physics.

Although the algorithm has an O(n2) complexity, it is much easier to parallelise than the more efficient variants. I found that my Ryzen 3700X CPU could run around 300 - 400 particles on a single thread, and around 1200 when utilising all of my threads.

Using this data, I decided that the maximum number physical objects in my game should be set to around 500, as this will leave plenty of room for other calculations and allow it to run on older machines.

Next I added collisions, because the objects are all spheres, collision detection is extremely simple. I then gave my objects mass and added this into my collision algorithm.

As objects gain mass, they need to grow in size. I started doing this linearly, however this is not realistic. I approximated how this would really work by making an objects radius the cube root of it's mass divided by pi.

Now I have objects that can collide with each other and exchange mass, I started making my spheres look more like asteroids. To do this I added into my vertex shader some fractal simplex noise, and changed the weightings of each octave to produce a nice looking asteroid.

Then to calculate the new normals, I added a geometry shader which calculates the normal for each triangle. Unfortunately this meant that the objects were flat shaded, as a geometry shader is unable to calculate the normals per vertex.

To overcome this I created a compute shader that added the noise to the vertices before they were rendered, then pre-calculated the normals which are fed into the shader stack.

At this point, I decided it was time to change my sphere representation from a UV sphere to an icosphere as a UV sphere has a higher vertex density at it's poles, and it would be best if I had an equal vertex density over the surface of my sphere.

To do this I first generate an icosahedron, and normalise it's vertices so that they lie on the unit sphere. Then I wrote my own algorithm which subdivides each triangle and normalises the added vertices onto the unit sphere, making sure not to duplicate any vertices.

At this point I started working on adding other levels. A level in code is an object which contains all of the game state, as well as a completion condition. When the completion condition of a level is met, the game engine destroys the level and fetches the next level from the level stack.

I decided to create a start menu, which allows the player to change difficulty settings and tweak the underlying physics constants to add replay value, the main menu is implemented as it's own level in the game code.

Throughout the development process I have been constantly refactoring the code to ensure readability and good code structure.

Code Structure

When designing the structure for this project, I decided to try and keep the design as extensible as possible, allowing for further development in the future to not have to catch up to significant technical debt.

In order to do this I decided to create several base classes which describe fundamental behaviours, and then extend these base classes to implement specific behaviours.

Below I will describe the core classes which make up the game engine.

Engine Class

This class controls starting the engine and the engine loop, and is where the game levels are stored

GameLevel Class

This class describes the common features between all levels, including setting up, running and destroying the level. The specific level classes extended these functions with specific elements required for each level.

RenderObject Class

This class describes the common features of any object that can be rendered to the screen, and includes functions for creating, rendering and destroying these objects.

PhysicsObject Class

This class describes the common features of any object that exists in the physical world, and includes functions for processing changes to it's physical properties.

GameObject Class

This class combines the RenderObject and PhysicsObject classes using multiple inheritance. I worked quite a lot on exploring how to do this, and after some experimentation decided that creating this class was the best way to combine the two behaviours.

Physics Class

This class provides functions to run the physics for an entire game world, including collision detection and gravity. In this project all objects are assumed to be perfectly spherical, which speeds up the collision calculation. I debated about creating a special instance for the asteroid level, where early on many objects are not perfectly spherical, but decided that with the time I have implementing this single special case would take too much time, and potentially be too resource intensive.

IcoSphere Class

This class contains functions that can create an icosphere, which is a sphere projection with a uniform distribution of vertices. I chose this projection because I am modifying the sphere vertex positions using a noise function.

GameSettings Class

This class stores static config settings which are set using the gui.

Utility Class

This class stores static functions which are used throughout other calculations for things such as rng.

Shader Class

This class loads shaders and has method to set shader uniforms.

Starting Point

I started this project already having what I believe to be a solid grasp on the fundamentals of C++, however I had not done much OpenGL programming in the last 5 years, so I referenced the OpenGL Red Book, as well as: https://learnopengl.com/.

For the lighting shader, I started with: https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/2.2. and modified it to suit my application, I also used https://github.com/ashima/webgl-noise/blob/master/src/noise3D.glsl for my simplex noise shaders, and https://github.com/ashima/webgl-noise/blob/master/src/cellular2D.glsl To generate cellular noise that is used in the background.

I would consider these noise functions to be library code.

The shader loading class is based off this class: https://learnopengl.com/code_viewer_gh.php?code=includes/l

The overwhelming majority of code used in this project is original work, including the code structure design.

Inspiration

Inspiration for this project comes from games such as Solar 2 and a similar flash game that game before it.

There are other similar games in the same genre, such as hole.io and agar.io, although I didn't pull from these for inspiration.

The main feature of my game is that it is rendered in 3d, as opposed to the 2d graphics of Solar 2. however Solar 2 has more game mechanics, which would have been unfeasible to implement in this time frame without leveraging a game engine such as Unity.

This project is more akin to a tech demo rather than a full game, due to it's lack of story, however this was considered during planning and I decided this was the best use of my time.