Interactive Drone in OpenGL and C++

1 Introduction

The deliverable software program produces a 3D simulation of a drone within a scene. Some parts of the simulated scene, as provided, are:

- A user-controlled drone object

  The drone object consists of a drone design, inspired by conceptual designs, featuring a spinning rotor which stops when the drone is landed at floor level. The drone acts as a point light source which affects everything else nearby but does not self-light the drone.

- A computer-controlled drone

  This drone is a wood-textured craft with a spinning rotor. Its movements are determined by the program. Holding the F4 key will disable the movements and cause it to return to the ground level.

- Movable objects (Person, Television prop, drones)

  Such objects can be collided with, and a resultant force will apply to the user-controlled drone and the movable object.

- Immovable (Static) objects

  Colliding with such objects as trees, rocks, buildings or the truck will impart a force to the moveable colliders but not the static object.

- Island terrain

  The island acts as a graphic to represent the floor level.

- Water

  A disk which uses a fragment shader to achieve a fluid animation of the surface

- Sky dome

  Three domes form the cloud, horizon colour and star layers of the sky. These are transformed by a hierarchy from the user-controlled drone so that they appear a constant distance from the viewer.

- Sun

  Point light source

2 Engine Code

The code is structured according to OO-principles of inheritance [TutorialsPoint, 2018]. Generic 'engine code' provides an API which can be extended for arranging and simulating any desired scenes. Some features of the generic code are:

- Sphere and OBB collision detection (and some initial code for OBB octree-to-octree leaf and triangle collision detection)

- Regular-grid spatial partitioning

- Model and shader reuse
- Bump mapping
- 4 lights per object
- UV Scrolling
- Multiple camera positions
- Debug visualisation

2.1 Implementation details

Some key decisions to consider when evaluating the source code are:

- The GameEngine class is an abstract class intended to allow calling code to be reusable with different subclasses.
- The GameEngine class contains an InputManager and a SceneManager instance.

  The InputManager provides routines for checking the state of keyboard and mouse input and features a per-key cooldown which are needed for toggle controls.

  The SceneManager provides routines to maintain multiple scenes and at runtime change which scene is the current scene.
- One such subclass is GameEngine3D, which implements GameEngine with general routines to expose functions for initialising, updating and drawing a given 3D scene.
- The Scene class is a basic description of a 3D Scene lists of lights, cameras and objects in the scene. It provides the code to draw and to maintain the scene.
- The GameObject class provides a representation of a 3D object position, rotation, scale, parent GameObject, physics attributes, transformations as well as references to the 3D model and point light source.

  A GameObject may represent a 3D model or a point light source by being added to either the lights list of the scene, the objects list or both.
- The Camera class extends the GameObject class so that cameras may be placed in GameObject hierarchies to inherit transformations from objects.
- Physics, Light and Material are simple classes to store and maintain attributes relating to physics, light sources and materials, respectively.
- Grid has a data structure to store GameObject references in a list corresponding to a spatial hash, as well as the routines to add GameObjects to the structure and regenerate the grid.
- DroneGameEngine is an extension of GameEngine3D class where the extended code initialises the simulation scene from the assets and extends the update routine for specific behaviour keyboard inputs and automatic behaviour.
- DroneGame1.cpp is the program entry point. This initialises an instance of GameEngine (specifically, DroneGameEngine) and manages the execution of the simulation in a while loop. This class is also responsible for determining and providing the time values to the

GameEngine. As an optimisation, an independent frequency is maintained for physics simulation (default 30 frames per second) and drawing (default 60 frames per second).

3 Dependencies and Assets

The engine code implementation relies on some embedded library code:

- glm/glew
- OBJ loader
- ThreeDModel
- Octree

The main simulation requires various assets, and the engine is capable of simulating scenes which utilise such assets as:

- Textured 3D models [1] stored in OBJ format along with associated MTL files
- Texture images (png bitmap files)
- GLSL Fragment and vertex shaders

4 Collision Detection

For collision responses, the primary concern is to determine whether a collision occurs for objects given geometry data, transformations and forces. A secondary concern is to determine all the points, or just the first point, where a collision occurs. The interactive drone application implements an algorithm using the separating-axis theorem to determine intersections.

The scene class iterates over each grid sector and, for each GameObject in the sector, *A*, invokes doCollisionsAndApplyForces with the list of all objects in the sector. For each object in the list, *B*, ignoring the invoked GameObject itself the function performs a sphere-to-sphere collision check between *A* and *B* with the largest extent of the bounding boxes as the radii. If the distance between the objects is less than, or equal to, the combined distance of the radii then there is an intersection. Depending on collisionType, the algorithm may return true as a sphereto-sphere collision is detected, or a subsequent test may be made in a call to doSAT to perform an OBB-to-OBB test. The doSAT function tests whether the oriented bounding boxes of two given 3D model are colliding using similar principles to [Bittle, 2010].

In the function doSAT, the axes to test are generated by obtaining world-space vectors representing 3 perpendicular edges in each box. [2] As per the separating-axis theorem, the algorithm generates a subsequent 9 axes from the cross product between the triplets. Iterating over each axis, the box vertices are projected onto the axis and the minimum and maximum values are determined. The projections are then tested for separation if the maximal point of one projection is greater than the minimal point of the other, there is no separation. If a separation is found, the algorithm quits immediately and returns false because there can be no collision if there

[1] The 3D models included have been created by myself using Blender 3D and are textured with my own textures or textures from cgtextures.com

[2] Only three are required per box since all other edges will be parallel to these.

is an axis of separation. Otherwise, if all axes are overlapping, the objects are colliding, and the algorithm returns true.

5 Spatial Decomposition

The regular grid structure, in the Grid class, contains a fixed-size 3D array of GameObject lists. Each array level corresponds to x, y and z sector coordinates.

Every frame, the Scene class creates a new grid and adds each GameObject to the grid. The sector coordinates for the GameObject are calculated with the following formula for the X, Y and Z coordinates: $(pos - (pos \% sectorSize))/sectorSize)$.

This is a relatively fast operation which provides large gains in performance by reducing the set of collision pairs to test by eliminating all far-apart[3] GameObjects from collision tests.

6 Evaluation

Considering problems with the implementation, some improvements which could be made in the future include;

- Visual quality in the fluid animation generated in the water fragment shader could be improved by implementing a better texture co-ordinate displacement factor in water.frag, such as the dynamic factor discussed by [Shehata, 2016], where the current factor is:

  ```
  coord.y += 0.02 * sin(time) * (sin(coord.y)); coord.x += 0.02 * cos(time) *
  (cos(coord.y));
  ```

- Occasionally, the magnitude of the collision response force is not sufficient to immediately prevent intersections. This occurs because other forces are applied as well as the collision response: an improvement would be negating other forces acting on the collider at the time of collision. For example, the drones acceleration due to user input could be ignored while a collision response is being applied. Currently, by applying both general forces at the same time as collision response forces, objects can resist the collision response force.

- Missed collision detection due to objects overlapping multiple grid sectors can be solved by naively checking objects in surrounding grid sectors. Alternatively, adding the object into all grid sectors where the geometry intersects rather than only the sector corresponding to the spatial hash of the object position. A third solution involves a specific arrangement of objects to avoid, or minimise the likelihood of, objects overlapping multiple grid sectors.

- Collision response activating when there is no geometry intersection can be improved by implementing triangle-to-triangle intersection tests to determine whether object geometry, rather than OBBs, are colliding. The attempted solution in the Collision class and can be viewed by choosing triangle from the collision detection options:

  ```
  GameObject::collisionType=0; //sphere
  GameObject::collisionType=1; //OBB
  GameObject::collisionType=2; //octree-leaf
  ```

[3] An issue addressed in this report is that sometimes collision tests between close or colliding objects are culled by such a naive regular grid implementation.

```
GameObject::collisionType=3; //triangle
```

References

[Bittle, 2010] Bittle, W. (2010). Sat (separating axis theorem).

[Shehata, 2016] Shehata, O. (2016). Using displacement shaders to create an underwater effect.

[TutorialsPoint, 2018] TutorialsPoint (2018). C++ inheritance.

A Screenshots

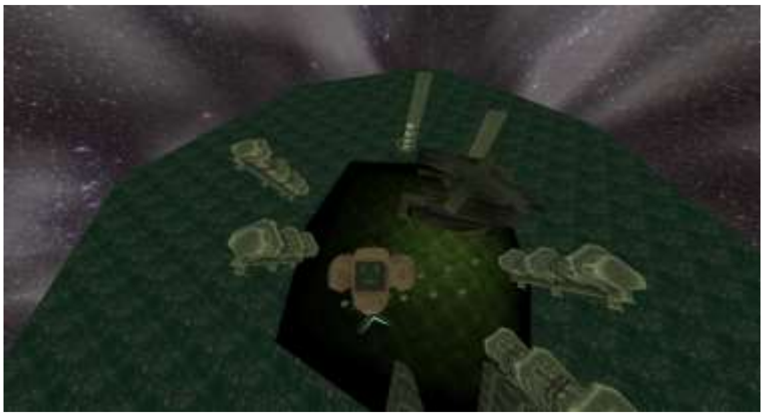

Figure 1: The user-controlled and computer-controlled drone flying above the scene

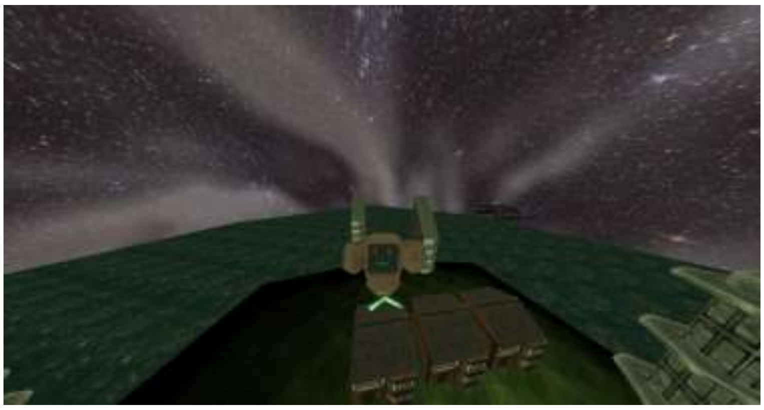

Figure 2: The clouds are animated using cumulative texture co-ordinate offsets

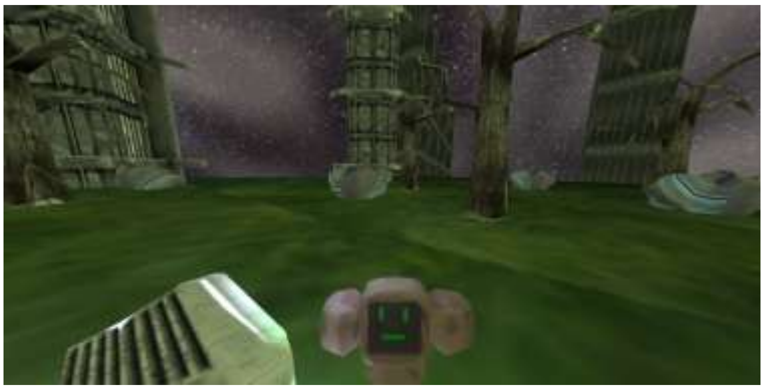

Figure 3: Specular light on the drone is emitted from the CRT television prop

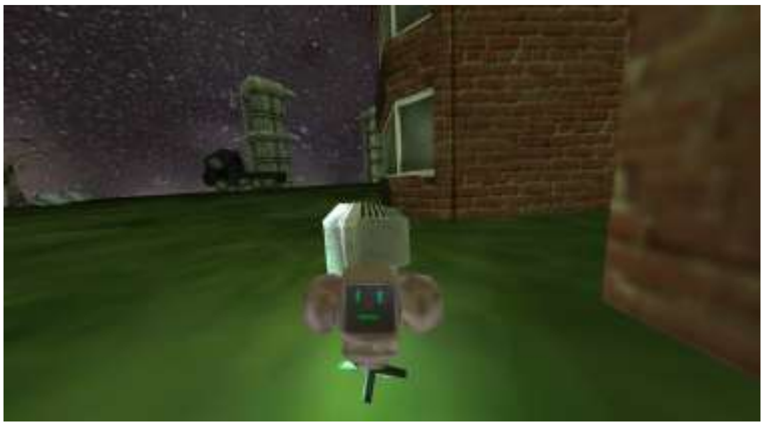

Figure 4: The CRT television can be moved by the user-controlled drone, effects of normalmapping can be noticed in the drone specular reflections

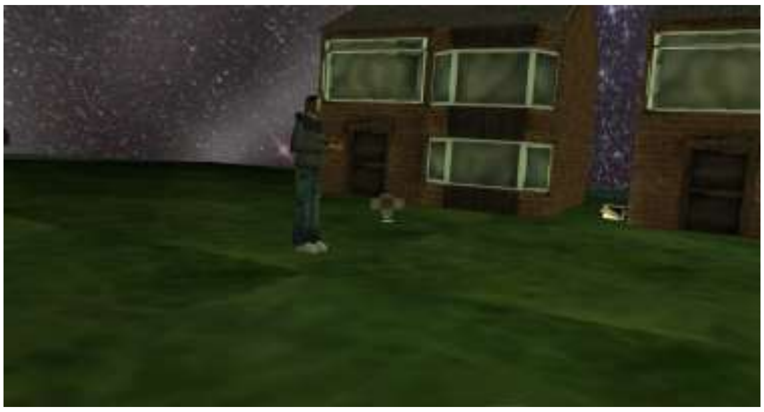

Figure 5: An alternative camera shows the drone controller character model

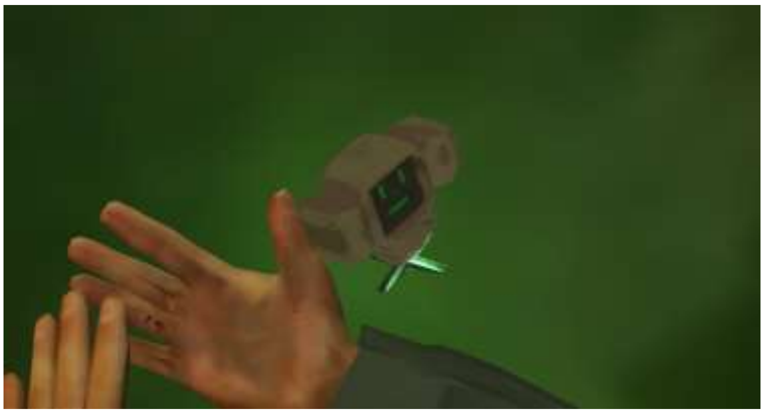

Figure 6: The view from drone controller perspective

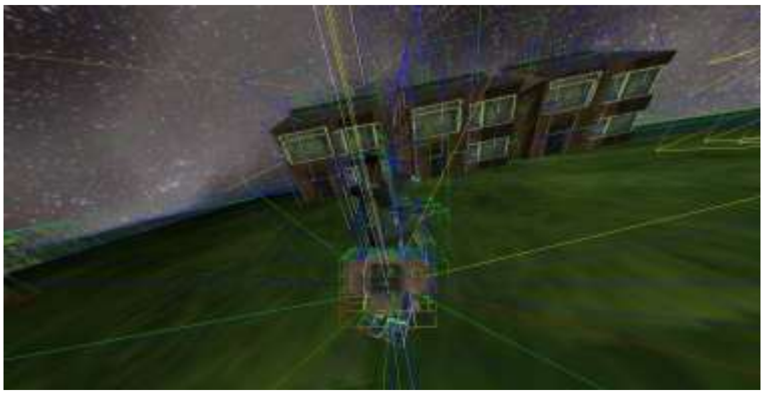

Figure 7: Debug visualisation shows bounding boxes and octrees

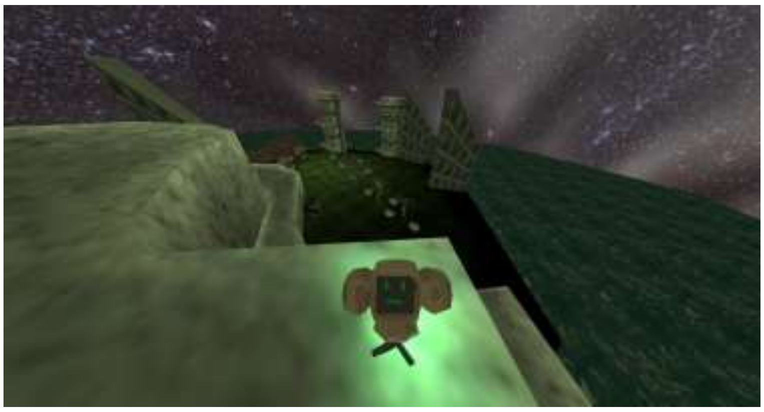

Figure 8: The user-controlled drone is a light source causing specular reflection on a building model