Computer Graphics 200 – Assignment 2

Declan White – 16142600

# Introduction

Attached is the source code to a simple interactive city generator for the Computer Graphics 200 OpenGL assignment.

Procedural generation is continuing to be significant within the computer graphics industry; hence I chose to base my work on that topic. To gain experience in game development field, there is an interactive nature to the program. Additionally, the code structure is inspired by the successful game engine “Source”.

Various external tools were used to assist in development, such as the AntTweakBar library which provides a convenient graphical parameter interface. Additionally, various models and textures were sourced from online resource directories.

# Program Features

Upon starting the program, the user is presented with a grid; this is used for the layout of the scene.  
Starting animation begins the procedural world generation, which involves placement of roads, structures and vehicles. The vehicles will begin navigating the road network and displaying lights.  
On the left of the screen is a GUI used for controlling the scene; additionally, the keys specified by the information text in the top-right of the screen can be used. The animation can be paused, resumed, quickened and slowed.  
The entire scene can be rotated.  
Various rendering components can be controlled, including lighting, wireframe polygons, accumulation, back-face culling, fog, and automatic normalisation.  
Restarting the animation will generate clear the current world and begin generation of a new one.  
The right mouse button can be used to move the camera around the scene, and keys *1* and *2* can be used to change the camera between top-down and car-following view.

# OpenGL graphics and animation

Although not strictly related to *my* campus, the scene chosen was to generate a random university campus; due to time and inspiration constraints, this was converted to a city scene.  
Within the scene there are number of simple and composite objects. The towers are composed of generated quads which are tessellated for increasing levels of detail required; the road corners are similarly generated procedurally using circle-segments. The statues and cars are loaded from external Wavefront Object files, which each consist of multiple shapes that must be rendered separately with different materials and OpenGL states (i.e: for transparency).  
Each object has a different combination of ambient, diffuse, specular and emissive surface finishes. Material Template Libraries are used alongside the OBJ files to specify the varying finishes for each model. The world grid is drawn using illuminated non-textured lines. Fog is enabled by default; linear fog was chosen for aesthetic reasons.  
Normalisation was not enabled by default due to the performance penalty on virtualised machines. Blending was used for transparency rather than accumulation due to incompatibly with virtualised machines.

# Code Design

The C++(0x) language was chosen for the use of extended feature set. Initially, the project was created in C, but later ported to C++11 to allow quicker development of the hierarchical entity class design, then C++0x to enable compiling at Curtin.

The hierarchical class design was inspired by its use in existing game engines, such as the Source and Unity engines.

## Class/File Roles

##### Main File

This contains the initialisation and bindings for GLUT, AntTweakBar and the CEngine singleton. Although the original idea was for all the code related to GLUT and AntTweakBar to be in the main file, much of leaked into CEngine and CRenderer due to time constraints.

##### Engine Class

This is the “main” class of the engine. It initialises the renderer, world and player and manages the entities for their lifetime. Events from GLUT are dispatched appropriately. It also provides a simpler interface to the AntTweakBar library, by allowing a variable to be adjust at any point in the code by adding one line.  
On every window display event, the frame is rendered and the delta time is accumulated in a variable. When the variable reaches a threshold, the simulation is advanced one “Tick”. This method is used to allow for smooth animation even in varying frame-rate. If the frame or tick takes more than 200ms to render, the simulation is paused to prevent the “spiral of death”. This algorithm was inspired by an article titled “Fix Your Time-step” (Glenn Fiedler, 2006). The state of the animation is control and set up according to the specification (A for play/reset, T for pause, etc.).

##### Renderer Class

The majority of the OpenGL work is performed within the single of this class, upon “window display” events being dispatched from the CEngine singleton. Each frame, the parameters for each feature is queried from the GUI and then the appropriate OpenGL calls are made to prepare the state. Some of the simple features are: wireframe, accumulation, back-face culling, fog, automatic normalisation, and global scene rotation.

The camera position, rotation and field of view is fetched from the CPlayer singleton and pushed to the appropriate matrices.  
Each entity’s absolute axis-aligned bounding box (AABB) is tested against a frustum to determine whether it is visible in the scene. If so, its distance to the camera is determined and a level of detail value is calculated (visible in the program by hovering over the entity). This level of detail is passed to the entity’s rendering function, which allows for tessellation and polygon-chopping (as discussed in the CProp and CPropBuilding information).  
Several render passes are performed for each entity:

* Lights: A pass allowing entities to add lights and material properties to the scene.
* Culled: A special pass for entities outside the frustum (the AABB is drawn to assist in debugging).
* Normal: The primary render pass, drawing every entity in the scene with textures, illumination, etc.
* Transparent: Entities with transparent elements are rendered and the result is blended onto the render buffer.
* Overlay: A special pass allowing entities to add info to the 2D debug text at the top right of the screen.

If the CPlayer singleton indicates the user is in the “top down” view mode, the nearest entity to the camera is made partially transparent to prevent it blocking the scene (and to satisfy the assignment criteria).  
After rendering the scene in perspective projection, an orthographic projection is loaded and the overlay pass is performed.  
After each frame, the OpenGL state is reverted by disabling any enabled features.

The CRenderer class provides some utility functions for each pass, such as light to add a light, and text to add text to the overlay. The pushEntity and popEntity functions are provided for the CEntity class to perform hierarchical rendering, though the depth does not exceed one in this demonstration.  
The loadTexture and findTexture functions are used in loading image resources for the CProp class. The function uploadTexture was intended to be used for procedurally generated textures, but the need did not arise.  
A cache of textures is kept to prevent duplication of resources and to hasten the loading process. The Simple OpenGL Image Loader library is used internally to load files and generate MIP maps. Appropriate functions are used to instruct OpenGL to use the MIP maps when filtering the textures.  
The screenToWorld function calculates the position and direction of a ray which represents a position on the user’s screen in the 3D world; this is used to discover which entity the user is hovering over, as discussed in the CPlayer information.  
The glutBitmapCharacter function is used to render text. The gluPerspective, gluOrtho2d, and gluLookAt functions are used to prepare the appropriate matrices.

Every call to a OpenGL, GLU or GLUT function is surrounded by the GL\_CHECK macro, which was originally sourced from a Stack Overflow reply (Mārtiņš Možeiko, 2012). It ensures correct usage of the APIs and prevents hidden bugs arising from misuse.

This is the only well documented class, due to time constraints.

##### Entity Class

All the common information for an object within the 3D scene is presented by this class. It provides useful utilities, such as the tween effect used in the building-rise effect, and handles the dispatching and state preparation of simulation and rendering events.  
Another important utility provided is the calculation of the AABB extents, which is used throughout the application.  
The virtual function calculateExtents is overridden by every subclass to ensure it correctly encompasses all the volume of the object and that culling and object picking operate correctly.

##### Player Class

This class handles the interface between the user and the 3D environment. All input events not handled by the GUI are dispatched to the singleton of this class.  
Keys for controlling camera orientation, global scene rotation, and world generation are hooked.  
Mouse movement and buttons are hooked for top-down camera movement and object picking (via the CRenderer screenToWorld function).  
There are two camera modes implemented:

* Top-down: A camera view from above the city, reminiscent of “Real-Time Strategy” games. By moving the camera lower and tilting it, you can fly through the city. (Shortcut: 2)
* FPS: A camera view locked to the nearest dynamic entity (a car). (Shortcut: 1)

The CPlayer class also defines most of the 2D debug text, namely the information on which keys control the demonstration.

##### World Class

Global world layout calculations are performed in this class.  
The state of the generator is initialised in initWorld, which prepare the uniform cell grid.  
Each call to initWorldStep advances the world generation by a fixed amount until completion. The CEngine class calls this repeatedly while the animation is running.

The generator is internally divided into stages, as you can see when running the program:

* Road Generation: A number of “road ants” are created in the centre and around the edge of the grid. Each step, these ants advance in a given direction and mark cells as being “road”. There is a random chance of the ant turning or branching left, right, or both. When an ant advances into an existing road cell, or the edge of the grid, it is marked as “complete.” Upon all ants completing, a CPropBuilding is placed on the cell’s position and instructed to generate a road-segment model depending on the surrounding roads. There is a random chance that a dynamic object (a car) will be placed on this segment.
* Lot Generation: The first empty cell next to a cell marked as “road” is found, and made the “active lot”. This lot grows rectangularly until either at a predefined maximum size, unable to grow in any axis, or given a random chance of early completion. Lots are repeatedly generated and grown until there are no empty cells next to cells marked as “road”.  
  After every lot is complete, they are iterated and a CPropBuilding is placed on each lot’s position and instructed to generate a building appropriate for the lot size. A tween animation is then cued to provide the aesthetic effect of the buildings “rising”.

Each cell holds the necessary information for roads and buildings, as discussed later.

There are numerous utility functions in the CWorld class to enable the exploration of the cell grid.

##### HeightField Class

This class is not used in the demonstration (due to time constraints).  
It is a grid of varying height, intended to be used for environmental terrain (such as a park or a hill).

##### Prop Class

This class is the base for all mesh objects; it provides rendering and extent calculation.  
The rendering function handles orientation, texture, material and transparency settings for each shape of the model

The functions gl\*Pointer and glDrawElements is used to perform the actual drawing. These were selected over the glVertex3f and associated functions for efficiency and convenience.

The CModel and associated utility classes are used internally, as discussed later.

##### PropBuilding Class

This class handles the generation of the various structures seen in the demonstration. Each instance is generated based on the properties passed to it by the CWorld singleton. It uses the grid utility functions to determine the road position and orient the structure appropriately. Other structure parameters, such as the mesh data, color and material are defined internally. There are a number of structure forms:

* Simple Building: A textured rectangular prism on a flat base. The textures are randomly selected for each shape (wall, roof, pavement). Despite the simple structure, it’s surprisingly convincing to the viewer.  
  Various levels of details are defined. The higher the level, the higher the tessellation of the quads. The increased number of triangles at close-range enhances lighting illumination. Decreasing it at a distance maintains program performance.
* Statue: One of many supplied model files are loaded and positioned in the centre of the structure’s area, along with a pavement base. The models supplied are: roman statue, teapot, windmill, observatory, rotunda, rabbit, court, Monumento Cristo; all of which were sourced from online resource directories and are credited below. A simple cube is generated for low levels of detail.
* Road: Specifically for road cells, this structure has different forms depending on the road situation, which are: Straight, Corner, T-section, Crossroads. Various levels of tessellation are generated for the levels of detail.

##### Car Class

This is the only dynamic entity class in the demonstration. It derives from the CProp class and loads a supplied vehicle model. In its simulation “Think” function, it analyses the road and makes decisions are appropriate. It utilises flag on each cell to mark the presence of vehicles to ensure they do not occupy the same space. Many hours were spent perfecting their behaviour. A simple cube is used for low levels of detail.  
If the car is a specific model (“copcar.obj”), the render function will insert red and blue flashing lights into the scene.

##### Model File

A set of utility classes and functions are provided by this class for model loading. The “Tiny OBJ loader” library is used internally to load Wavefront Object files and associated material libraries.  
Utility functions are provided for generation of models: pushVertex, pushQuad, makeCube.  
Originally, utility functions for live polygon reduction (via the “polychop” algorithm) were included, but have been omitted due to time constraints.

##### Matrix Header File

This file contains the various math and data classes needed for working with the 3D environment. The classes were personally implemented, but the majority of methods sourced from online resources, as credited below.  
The utility classes are: mat4, vec3, vec4, quat, ang3, col3, col4, line, plane, triangle, box\_aabb, and frustum.

# References

<http://www.glprogramming.com/red/>

Information on OpenGL usage.

[www.tf3dm.com](http://www.tf3dm.com)

Wavefront Object files used in this demonstration.

<http://www.rockthe3d.com/47-free-high-quality-building-textures/>

Building textures using in this demonstration.

<http://www.assistcg.com/component/image/category/61.html>

Building textures using in this demonstration.

<http://en.wikibooks.org/wiki/OpenGL_Programming/Transparency>

Information on transparency and blending in OpenGL.

<http://stackoverflow.com/a/11258474/837856>

OpenGL error checking macro for C++.

<http://stackoverflow.com/a/2670913/837856>

Token to string macro for C++.

<https://github.com/rbarril75/Scratched-Blue-Teapot>

Model and texture for blue teapot.

<https://github.com/ratkins/RGBConverter/blob/master/RGBConverter.cpp>

HSL to RGB conversion function for C++

<http://gafferongames.com/game-physics/fix-your-timestep/>

Information on physics simulation timing.

<http://stackoverflow.com/a/15813530/837856>

Filtering non-printable characters in C++.

<https://github.com/zeyangl/MassLight/blob/master/src/Mesh.cpp>

Estimating vertex normals for a mesh with none specified.

<http://stackoverflow.com/a/7596981/837856>

Calculating the inverse of a quaternion.

<https://github.com/ValveSoftware/source-sdk-2013/blob/master/mp/src/mathlib/vmatrix.cpp>

Code for 4x4 matrix multiplication.

<http://www.arcsynthesis.org/gltut/Positioning/Tut08%20Interpolation.html>

Information on vector interpolation.

<http://mathproofs.blogspot.com.au/2005/05/uniformly-distributed-random-unit.html>

Information on uniformly distributed random quaternion generation.

<http://willperone.net/Code/quaternion.php>

Code for quaternion and Euler angle conversion.

<http://pastebin.com/PkVzFkf4>

Code for spherical quaternion interpolation.

<https://bitbucket.org/mogre/mogre/src/4327903e1bed/Main/src/Custom/MogreVector3.cpp>

Code for quaternion rotation calculation given two normals.

<http://math.stackexchange.com/a/84088>

Information on line-plane intersection.

<http://www.geometrictools.com/LibMathematics/Intersection/Intersection.html>

Information on line-triangle intersection.

<http://gamedev.stackexchange.com/a/18459>

Information on line-AABB intersection.

<http://zach.in.tu-clausthal.de/teaching/cg_literatur/lighthouse3d_view_frustum_culling/index.html>

Information on frustum-related material.