MacPac

Game Programming

Assignment 2

Name: Stuart Bryson
Student #: 98082365

MacPac was written for assignment 2 of Game Programming at UTS. Building on assignment 1, it has the following new features. Axis aligned bounding boxes, Quad/OctTree, notifications, further drawing options and a maze generator.

Features

Axis Aligned Bounding Boxes

Axis aligned bounding boxes are a very cheap way to test objects for rendering or collisions. Unfortunately, AABBs can be fairly inaccurate when surrounding sparse objects or objects which are not aligned with the axes. For this assignment however, the maze that is generated lends its self extremely well to AABBs.
The AABB class I have created allows the AABB to be extended by adding points or other aabbs. It also allows transformations and translations on the AABB points. It is important to note that while an AABB usually specifies 2 points in 3D space, a min and max, to transform the AABB, you cannot simply transform the min and max points. If you simply transform the min and max, it will result in an AABB that is either not axis aligned or rather, is no longer correctly containing the included objects in the AABB. Instead, to transform the AABB, the 'virtual' points of the AABB must be transformed and then a new AABB must be calculated by containing the 8 transformed points. For this reason, we also have a translate method which allows us to translate the min and max points directly, By simply translating the points we still retain an AABB.
The AABBs are calculated per Scene Entity and are then used to put Scene Entities into their appropriate node in the Quad/OctTree.

Quad/OctTree

An OctTree recursively divides the world into 8 equally sized boxes. It will continue to divide each of these boxes into another further 8 boxes until the desired depth is reached. Most trees will only store objects in the leaves of the tree and any objects that intersect multiple nodes will be stored in all the nodes with a flag to determine if the object has been rendered this frame. I decided not to store intersecting objects in multiple leaf nodes but rather store the object only once in the lowest node in the tree that fully contains the object. This means that I may have some objects in the top level node and others may be 2 levels deep. Storing objects in non-leaf nodes allowed me to treat all OctTree nodes the same and removed the complexity of keeping track of objects in multiple nodes and setting and checking rendering flags.
Once I had my OctTree working it became quickly apparent that an OctTree is of very

little benefit to my generated maze that was only 1 level high. The reason for this is due to the vertical subdivision that happens during the first subdivision of the OctTree. As all the walls in my maze extended the full height of the first level of my OctTree, the first subdivision would exclude these nodes and consequently all walls in my maze would be stored in the top level node. This of course was defeating the whole point of the tree.
It was then obvious that I should not be making that vertical division and therefore I should be using a QuadTree for the maze. Using the QuadTree resulted in the maze objects in the maze slotting into many different depths of the tree.
I then came up with the idea of making my tree adaptively decide between an OctTree and a QuadTree at each depth of the tree. If the height of the current node is less than half the width or depth then we only create 4 children of that node. Conversely we create 8 children when the node is taller. This technique could be extended even further to create quads or simply binary divisions in the most appropriate directions based on the dimensions of the current node.

The Quad/OctTree is used for both rendering and broadphase collision. The rendering tests the bounding box of the top level node in the tree against the openGL view frustum. If the node is visible, any drawables in that node is drawn and the child nodes are recursively tested against the frustum and they too are drawn. If any node in the tree is not visible in the frustum, it is not drawn and none of its children are even tested. This allows extremely efficient frustum culling of the scene. It does not give us Occlusion culling which significantly improves performance, however, generally occlusion culling is difficult nowadays as most games support semi-transparency in openGL shaders.

I also use the OctTree for broadphase collision detection. As the camera moves around the maze, I test the cameras bounding box against the octTree. For each node in the octTree, I then do a midphase detection using the bounding box of each entity. At this stage I have not implemented narrow-phase collision detection. BSP trees allow for very simple narrow phase collision tests. As I chose not to use BSP trees for this assignment, the amount of work required to implement narrow phase collision detection was not feasible.

Entities

Entities now include an AABB. As mentioned previously, this AABB is used by the OctTree for both rendering and midphase collision detection. The AABB is stored as both a local object space AABB and also as a world space AABB. Every time the entity is transformed, the world space AABB is calculated using the transformation and the local space AABB. Enabling the update of Scene Entity transformations and AABBs prepares the engine for dynamic or active objects.

Further Drawing Options

In implementing my OctTree, the bounding boxes, both in local and calculating the world space BB, and just in debugging objects, it became apparent that I needed to be able to visualize exactly what was happening. For this reason I implemented the following drawing features:

- The ability to draw in wireframe mode

- Drawing of local space AABBs. It does this by relying on openGL to concatenate the parent matricies such that the drawing of the local AABB will be drawn in direct correspondence with the actual geometry
- Drawing of world space AABBs. As the OctTree works with the world space AABBs and not the local space AABBs, we needed to calculate and ensure that the world space AABBs were correct. By visualising it we can compare it to the local space AABB.
- Drawing of the OctTree. Not only can we draw the OctTree outline, but we can also draw the OctTree at each depth. This allows us to visualise which objects have been added to which OctTree nodes.
- We can also toggle on and off the drawing of the objects using the OctTree and the drawing of the OctTree while testing the nodes against the frustum. Using these methods, we can check our object counts to see how efficient our OctTree is.

Maze

One of the requirements of Assignment 2 was to procedurally generate a maze. The maze is generated using the simple algorithm demonstrated in class. A few nice additions include the indication of the start and end cells in the maze, randomly placed walls that contain windows to allow for a more interesting experience, and a camera spline that gets generated when the maze is. When entering demo mode after a maze has been generated, the camera will follow the automatically generated spline that will take the player through the maze from the start to the finish.

Statistics Draw Info

A very useful feature is the ability to display statistics on the engine in the openGL window. Useful stats that I draw are the current camera position, current object count, and also any commands that are issued by the user such as toggling wireframe. These stats can be extended to include important information such as FPS, memory usage and more.
To implement the stats drawing, I used some code from Apple that allows me to create an openGL texture from a string and then draw this texture on screen. This is extremely efficient and it is also aesthetically pleasing.

GUI

I have further updated the GUI for this assignment. I have added menu items for all the drawing options mentioned previously. I have also added more preferences including the ability to specify the maze size and the octree depth. All of these options are saved in the users preferences and hold their state across instances of running the application.

Notifications

Another feature is the implementation of notifications. Notifications are a great way to message between objects for non-critical updates. Currently, I only use notifications for scene information messages that appear to the user when draw info is turned on. These notifications can be called from any object whether they are in our model, view or controller constructs. Examples of these messages include the toggling of wireframe on and off and the generation of a maze.

New Controls

Lastly, new controls were implemented to accommodate the new requirements for moving around the scene. The last assignment used a trackball implementation from Apple. This trackball was really useful for viewing a single object but is not appropriate for navigating a complex scene such as a maze. For this reason new controls were implemented similar to your typical first person shooter.
The implementations of these controls are done using the NSView class’s implementation of user events such as keyDown. This class is not well suited to game control interaction. In fact Apple recommend using the HID framework that is very extensive including support for force feedback. This API is however a steep learning curve and the NSView’s user input was much easier to tackle. This does however mean that I have not implemented mouse viewing due to a limitation in NSViews implementation. Following are the new controls:

Forward:	8
Back:	5
Turn Left:	4
Turn Right:	6
Strafe Left:	Z
Strafe Right:	X
Up:	D
Down:	C

The up and Down Arrows display different depths of the OctTree.