CS 297 REPORT

Optimal Meeting Point in a Game Framework

Nithin Reddy
Spring 2006
Advised by Professor Chris Pollett
Department of Computer Science
San Jose State University

ABSTRACT

This report is a detailed summary of the the first semester of San Jose State University's Computer Science Master's Writing Project. Included in the report is some background on the topic of study: Robot Meeting Point Algorithms. Following the background information are rundowns for each of the four deliverables that were developed during the semester. Lastly we outline the work to be accomplished in the following semester. A robot meeting point is useful when there are robots are distributed in some environment and need to come together to get work done. Robot meeting point algorithms have existed for decades, however many have focused on solving the problem in a planar world. A group from Carleton University in Ottawa, Canada have developed an algorithm that efficiently solves the meeting point problem for robots scattered on 3d terrain. The goal of the project is to develop a game framework that will allow us to analyze the efficiency of this algorithm.

TABLE OF CONTENTS

Introduction

The purpose of the first semester of the writing project is to begin research on the topic of study. The traditional reading portion of the research is complemented by implementation. The reading and implementation culminated into four milestones, or deliverables. The first deliverable involved having a flying saucer moving around a black background. The second deliverable was a remake of the classic Pick N Pop. The third deliverable explored the details of meshes. The fourth deliverable implements a planar meeting point algorithm.

This project will implement the shortest path meeting point algorithm of multiple robots on a weighted terrain into a game framework. The game framework will then be used to study the algorithm in an attempt to verify the algorithm's efficiency. The research project will also examine some aspects that the original algorithm disregards, such as the impact of different types of robots and different terrain types.

The game framework will be based on a modified version of ex-San Jose State Professor Rudy Rucker's POP framework. In the first phase of the project the POP framework will be ported to C# 2.0, and will use managed DirectX. The purpose of porting the POP framework is to simplify the existing code base. New packages will be added to the framework to support height-based terrains, as the current POP framework only supports flat terrains.

Deliverable 1: Taking Off with a Flying Saucer

Motivation

The first step when starting a project is to successfully setup the development environment. The next step is to produce a working demo.

Goals

The goal of this deliverable was to produce a Direct3D demo of a UFO moving around the screen. During the development of the demo the author should have setup Visual Studio .NET and DirectX, and gained some familiarity with these development tools.

Implementation and Results

A DirectX game uses a Windows Form to run in. The typical Windows Form application waits for events (such as keystrokes and mouse clicks) before taking any action. Once an event is registered, the application then responds by processing the event. However, this behavior does not apply to a DirectX game.

A DirectX game runs in a constant, tight loop. During each iteration of the loop several things happen. The scene is graphically rendered, user interaction from input devices is processed, and some game logic is executed.

In recent years graphics has become the most important of games. When a game is first previewed to audiences, the visual presentation of the game is what makes a lasting impression. Initial excitement for a game is almost always based solely on how it looks. Because of this, game developers want to continually raise the graphical bar with each new generation of games. The rapidly evolving capabilities of graphics cards have allowed game developers to do so. Direct3D, the graphical component of DirectX, exposes the capabilities of the graphics cards to game developers.

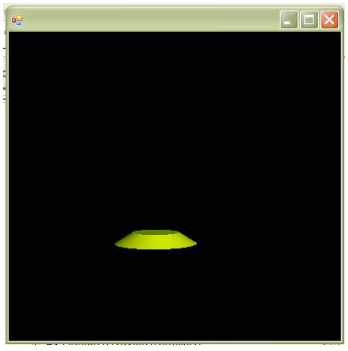
A Direct3D device represents a graphics card. It is a global object, so it passed around from object to object like a village bicycle. The device is a state machine, meaning that once

something is set, it remains set until changed. This can be a pitfall, because an object that receives the device has no idea what state the device is in.

Like all parts of the game loop graphics must be updated on each iteration. Within this update Direct3D has another loop, which includes the following steps:

1. Setup the world, view, projection and camera matrices
2. Setup lighting
3. Setup the device
4. Draw
5. Cleanup

This demo created a basic implementation of a Direct3D scene (shown below). A modified cylinder mesh represents the flying saucer. There are two lights in the scene: a directional light and a point light. The scene was animated using a simple path loop, which was fed the elapsed time between calls.



A material was given to the flying saucer to show off the lighting. Movement of the flying saucer was achieved by multiplying the transformation matrix with the world matrix.

Remarks

All the goals of this deliverable were achieved. Setting up a DirectX project was not very difficult, especially given the amount of resources available on the Internet. The only drawback is that since so much is automatically taken care of, it is difficult to understand what's going on behind the scenes.

Deliverable 2: Pick N Pop

Motivation

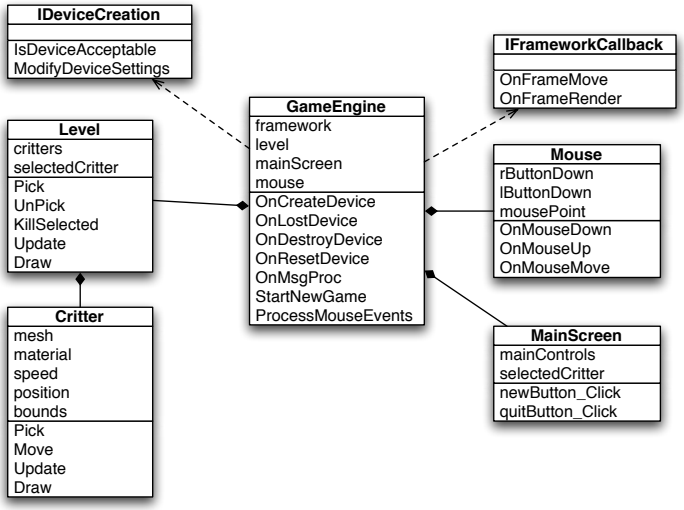
One of the overall goals of this project is to port Professor Rudy Rucker's POP framework to a new platform. In doing so the parts of the framework that are well designed should be kept, while the parts that are either outdated or unnecessary should be removed. The problem comes when trying to figure out what parts of the framework should be kept or thrown out. Instead of spending wasting time in a design phase that may or may not end in the right decision, we can take a more direct approach. We can create a game and add parts of the POP framework as needed. Once the game is done we can extract the distilled version of the POP framework used in the game as a starting point to port the entire framework.

Goals

One goal of Deliverable 2 is to recreate the Pick N Pop game on a new platform. Pick N Pop was chosen is because it is the namesake of the POP framework. It was the first game that Professor Rucker created, and the game from which he extracted the POP framework. So it seemed suitable that the new POP framework would be extracted from a new version of Pick N Pop.

Another goal of this deliverable was to learn a new aspect of DirectX, DirectInput. This provides support for input devices such as mice, keyboards, and joysticks. Capturing and processing user input is the last aspect on the path to creating a basic game. Other things, such as networking and sound, while important, are not necessary at this stage.

Implementation and Results



This demo shows shows the basics of a Direct3D game. I implemented a basic game loop consisting of update and render steps. The update step is used to check the state of the world and change the state based on time and user interaction. The render step draws the scene. There are five major classes: GameEngine, Level, Critter, MainScreen, and Mouse (see figure above). The GameEngine is responsible for initializing and handling Direct3D, DirectInput, User Interface, and overall game logic.

The Level class maintains the state of the current level, including: time remaining, score, the popcorn objects, and the gem objects. It also contains methods for manipulating the state such as selecting a critter, deselecting a critter, and killing the selected critter. Finally a Level object can determine what state it should be in for the player to win or to lose the game. For example, if time runs out the player has lost the game.

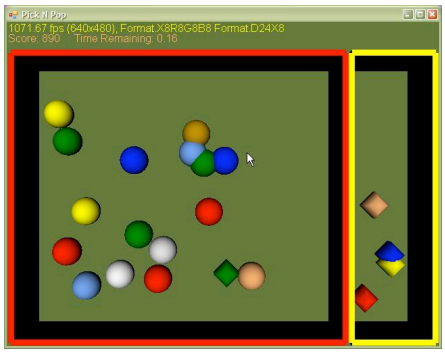
The Critter class contains information and methods to manipulate critters in the world. The Popcorn and Gem classes inherit from this class (not shown in above figure). Each of the derived classes contains a static mesh, which is used by all instances of the class. The alternative, creating a mesh for each instance, would consume memory proportional to the amount of critters in the world.

The MainScreen contains the GUI for the program (shown in image below). One of the shortcomings of the POP framework is that it relied on standard Windows controls to provide a GUI. The problem with this is that it gave a poor user experience when in full screen mode. When POP was running a game in full screen mode, you were not able to get to the controls that allowed you to set the options or quit the game. Having the GUI controls within the game provides the user with a seamless experience.

DirectInput provides a familiar type of access to all input devices. Back again is the Device concept used in Direct3D, where there is a global device for a particular piece of hardware that is shared amongst objects in the game. There was a problem when using DirectInput to interface with the mouse. There are two modes that DirectInput reports mouse position in. The first mode gives the relative position of the mouse to its last position. This mode seems to be designed for first person shooters, where relative position would indicate how far to move the camera between updates. The second mode gives an absolute value. The problem with this mode is that the absolute value is not relative to the window or to the screen. I believe this mode is used when you are creating a custom mouse for your game, but it is quite useless when using the windows mouse. So instead I used the standard windows mouse event to capture mouse events.

The other classes in the demo serve ancillary functions, such as storing the state of the camera.

The purpose of the game is to move all of the gems, represented by diamonds, into the yellow box (see image below) before time runs out. If you have trouble picking a gem, because of all the popcorn (spheres) moving at the same time, you can pop some popcorn by right-clicking on them. There is a danger to this, because you can also pop gems, which results in negative points. You win when all of the gems are in the yellow box, and you lose if you pop all of the gems or if time runs out.



Remarks

The new version of Pick N Pop uses a small subset of the original POP framework. As subsequent games are developed more, but not all, of the POP framework will be ported. Standard device input event handling was used because of the problems with DirectInput. The limitation of using standard Windows events is that supporting joysticks will not be possible. I think the issues with the mouse position can be overcome by using a custom mouse cursor. This would also give open up the opportunity to control where the mouse is. For example, we could move the mouse to a position in the scene to point out game mechanics during a tutorial.

Deliverable 3: Meshes

Motivation

In order to create a game world, there needs to be game-world objects which model real-world objects. These game-world objects should allow us to visualize the real-world objects that they represent. In a 2D game world these objects were usually sprites, whereas in a 3D

game world the objects are meshes. From animated objects to obstacles, meshes are used to graphically represent almost every real-world object. The detail of the mesh influences how realistically the representation appears. Since meshes are a key element to games, it is important to learn how they work.

Goals

The goals of this deliverable were to learn how to create a mesh and to understand the structure of a mesh. Left as they are, both of these goals are fairly broad. For example, creating a mesh also includes learning how to use a 3D modeling program. To narrow the scope of the goals only the basics of each goal is explored in this deliverable.

Implementation and Results

There are four ways to create a mesh object:

1. Load the mesh from a file (.x format)
2. Clone the mesh
3. Use a shape-creation function
4. Use the mesh constructor

This deliverable implemented two of the methods of mesh creation: it shows a mesh loaded from a file and a mesh created by using the mesh constructor.

Loading the Mesh From a File

The first method of creating the mesh is by far the most popular. Meshes are graphical representations (models) of real-world objects. Things like trains, cars, people, silverware can all be represented by meshes. Creating a detailed model can be difficult, so they are usually made with the help of modeling programs such as 3d Studio Max. The models generated by

the modeling programs can be converted into a .x file, the built-in mesh format for DirectX. The conversion process is handled by a utility program or plugin.

To load a DirectX mesh from a file, Direct3DX provides FromFile static method in the Mesh class.

```
public static Mesh FromFile(
      string filename,                // location of DirectX Mesh file (.x format)
      MeshFlags options,              // options for loading the file
      Device device,                  // Direct3D device
      out ExtendedMaterial materials  // materials and textures
);
```

A typical FromFile call looks like this:

```
Mesh mesh = Mesh.FromFile("tiger.x", MeshFlags.SystemMemory, device, out exMaterials);
```

MeshFlags.SystemMemory indicates that the mesh should be loaded into main memory and not into a page file. The exMaterials array will be filled ExtendedMaterial objects which describe the materials and textures used by the mesh. The mesh itself is divided into subsets; each of the subsets specifies its own material and texture.

Direct3D cannot directly use the ExtendedMaterial object, so we split it into objects that Direct3D can use: Materials and Textures.

```
textures = new Texture[exMaterials.Length];
materials = new Material[exMaterials.Length];
for (int i = 0; i < exMaterials.Length; ++i)
{
  if (exMaterials[i].TextureFilename != null)
    textures[i] = TextureLoader.FromFile(device, exMaterials[i].TextureFilename);

  materials[i] = exMaterials[i].Material3D;
}
```

We now have enough information to render the mesh.

```
for (int i = 0; i < materials.Length; ++i)
```

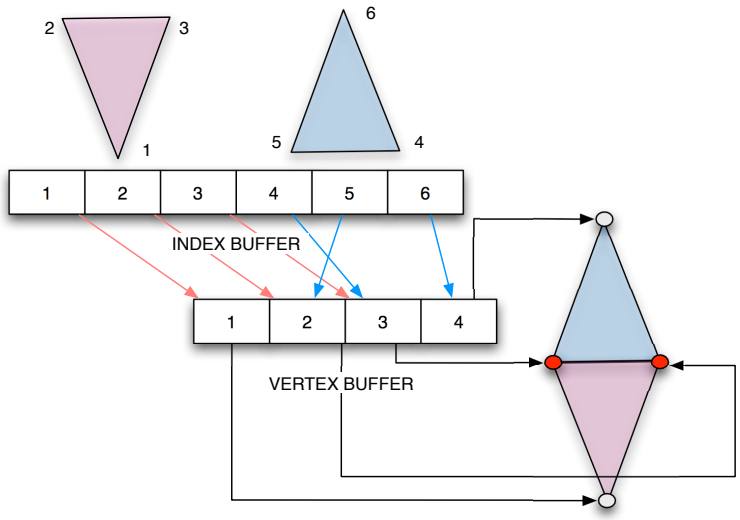
```
{
    if (textures[i] != null)
      device.SetTexture(0, textures[i]);

    device.Material = materials[i];
    mesh.DrawSubset(i);
}
```

Using the Mesh Constructor

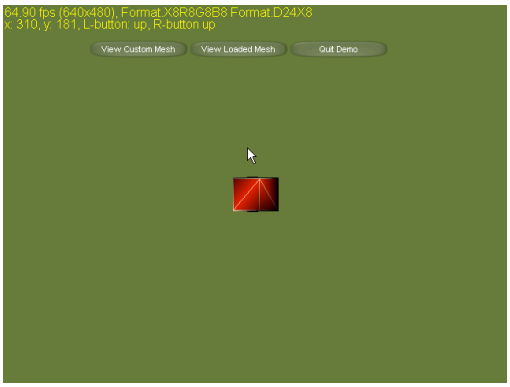
This method of creating a mesh is more difficult, because you must specify all the vertices of the mesh manually. The level of difficulty depends on the complexity of the mesh. Hand coding a cube mesh would be easier than creating a table mesh, which, in turn, would be far simpler than manually generating a robotic arm. Despite the difficulty there are situations when you won't be able to use a mesh fashioned in a modeling program. For example, you would need to generate meshes on-the-fly if you are imaging the bottom of the sea in real-time.

Besides materials and textures, meshes are made up of triangles. The reason for describing a mesh using triangles, is that graphics cards are optimized to process triangles. A set of three vertices defines a triangle. A vertex buffer stores all of the vertices that make up a mesh, while an index buffer joins sets of three vertices (i.e. defining a triangle). Using an index buffer reduces the storage requirements by eliminating vertices shared by two or more triangles from being listed twice in the vertex buffer.



Looking at the figure above we see that the two triangles share vertices 2 and 3 (shown in red). The first three elements of the index buffer define what vertices makeup the pink triangle, and the last three elements makeup the blue triangle. Without an index buffer, the triangles would be defined by six vertices. With it, the two triangles would only need four vertices combined to define them. Note that this only applies to vertices that are exactly the same. If the vertices have different normals, then they can't be shared.

In Deliverable 3, the demo program uses the same critter to render two meshes: a mesh loaded from a file, and a custom mesh. The mesh loaded from a file is the default mesh that is displayed when the program starts (left image below). To switch to the custom mesh, press the "View Custom Mesh" button. The custom mesh is shown then replaces the file-loaded mesh on the critter (right image below).



The custom mesh is a cube. The cube is made up of 12 triangles (two for each face of the cube). Each of those triangles has three vertices, so we would have 36 vertices in our vertex buffer without using an index buffer. A cube has eight distinct vertices that are share amongst the six faces, so our vertex buffer contains eight vertices. Once we define both our vertex buffer and index buffer, we can create the mesh like so:

```
customMesh = new Mesh(12, 8, 0, VertexFormats.PositionNormal, dev);
customMesh.SetVertexBufferData(vertices, LockFlags.None);
customMesh.SetIndexBufferData(indices, LockFlags.None);
```

Remarks

The program successfully demonstrated how to create a mesh in two different ways: using a pre-rendered mesh loaded from a file, and by defining a custom mesh. As mentioned in the Goals section there are more advanced mesh topics that were not covered in this deliverable. Two topics of particular interest are mesh animation, and creating a mesh hierarchy. Mesh animation allows the mesh to move using linear interpolation and key frames. A mesh

hierarchy is related to animation by defining how different subsets of the mesh interact with each other.

Deliverable 4: Planar Meeting Point

Motivation

The focus of this project is meeting point algorithms; more specifically meeting point algorithms in weighted terrain. It seems prudent to first start off with the meeting point algorithms for planar surfaces, before moving onto weighted (or 3D) terrain. The meeting point problem is also known as the facility location problem, because of the most common example given when describing the problem. The example goes, there is a shared facility (like a hospital or post office) that will be built in a community. The goal is to build the shared facility in a location that minimizes the distance to the furthest household in the community. The solution for this problem is the center of a minimum enclosing circle. A minimum enclosing circle is the smallest circle that contains all of the households (points). There are several solutions to this problem, that range in complexity from $O(n^4)$ to $O(n)$. For example, Elzinga and Hearn published an $O(n^2)$ algorithm in 1972, and Megiddo showed that a solution could be found in $O(n)$ time in 1983.

Goals

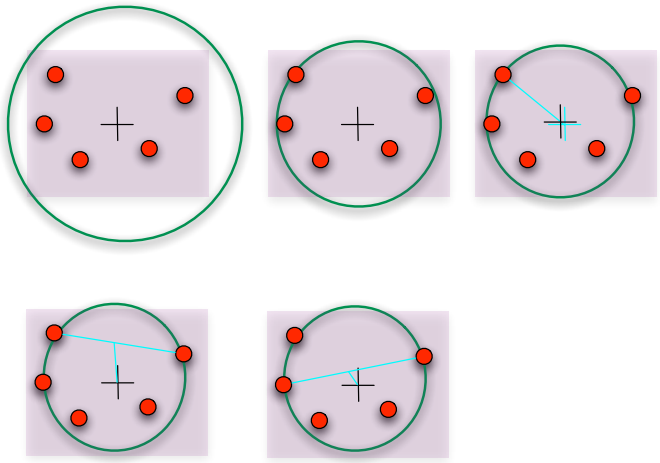
The goal of Deliverable 4 is to implement an algorithm to determine the optimal meeting point of robots in a plane by finding the smallest enclosing circle. We would also like to animate the solution so that it is easier to visualize. When deciding which of the algorithms

to implement, the priority was to choose the algorithm that can by accompanied by graphics at each step.

Implementation and Results

The algorithm that was chosen has four steps:

1. Draw a circle that enclose the borders of the world. This ensures that all points are enclosed by the circle, since they cannot be beyond the borders of the world.
2. Find the point furthest away from the center, C, of the circle, call the furthest point P. Draw a new circle with the same center that passes through the point P.
3. If the passes through two or more points, skip to step 4. Otherwise shrink the radius along the line segment PC until the circle passes through a second point.
4. Measure the arc length of the circle between all points. If the circle contains an arc length that is greater than half the circle's circumference, then the circle can be made smaller. Let's call the end points of this large arc length Q and R. While keeping Q and R on the boundary of the Circle reduce the diameter of the circle until either
    a. The diameter is the distance QR (you are finished), or
    b. The circle touches another point, S. In this case we repeat step 4.



In my implementation I was only able to get past the third step. I was able to create a system of two geometric equations for finding the center and radius along a particular line that allows the circle to pass through two points. However, solving this system for one variable turned out to be a nasty algebraic equation. Coding and debugging the equation on the computer proved very time consuming, and I ran out of time to implement the remaining steps of the algorithm.

Remarks

This deliverable was the first step toward implementing meeting point algorithms. However, due to time constraints I was not able to complete the implementation. The next step is to finish the implementation of this algorithm. Afterwards, it is important to also implement at least one other planar algorithm. The reason this is important, is that it allows us to create an algorithm testing module to POP, which can be used to later explore the weighted terrain algorithms.

Conclusion

In this semester we laid the foundation to continue our project next semester. We have started porting the POP framework to the DirectX / C# platform. New features, such as a in-game GUI, have been added to the POP framework. Finally, we began implementation of meeting point algorithms.

In the next semester the new POP framework would need to add two key features: performance testing of meeting point algorithms and height mapped terrains. We would also need to implement several meeting point algorithms for planar and 3D terrains.

BIBLIOGRAPHY

M. Lanthier, D. Nussbaum, and T-J Wang, "Calculating the Meeting Point of Scattered Robots on Weighted Terrain Surfaces"

R. Rucker, "Software Engineering and Computer Games." Addison Wesley, 2002.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to Algorithms." MIT Press, 2001.

Andera, Craig. Mesh Basics Tutorial. Retrieved May 8, 2006, from http://pluralsight.com/wiki/default.aspx/Craig.DirectX/MeshBasicsTutorial.html

Andera, Craig. Mesh Creation Tutorial. Retrieved May 8, 2006, from http://pluralsight.com/wiki/default.aspx/Craig.DirectX/MeshCreationTutorial.html

MSDN DirectX Documentation from the February 2006 SDK.

Schuld, Michael. (2005, June 17). Tutorial 5 :: Rendering Full Textured Meshes. Retrieved May, 2006, from http://www.thehazymind.com/archives/2005/06/tutorial_5_rend.html

Miller, Tom. Beginning 3D Game Programming. Sams Publishing: Indianapolis, 2005.

Pierson, Derek. (2005). Beginning Game Development Part V - Adding Units. Retrieved February, 2006, from http://msdn.microsoft.com/coding4fun/gamedevelopment/beginning5/default.aspx

Eliosoff, Jacob, Unger, Richard. (1998) Welcome to the Minimum Enclosing Circle Emporium. Retrieved May, 2006, from http://www.cs.mcgill.ca/~cs507/projects/1998/jacob/

Muhammad, Rashid Bin. Smallest Enclosing Circle Problem. Retrieved May, 2006, from http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/Center/centercli.htm

D. J. Elzinga and D. W. Hearn. Geometrical solutions for some minimax location problems. Transportation Science, 6(4):379--394, 1972.

Megiddo, N., "Linear-time algorithms for linear programming in R3 and related problems". SIAM Journal on Computing, vol 12 number 4, Nov 1983.