Hoki Hoki Source Notes

Max Abernethy

2006-01-28

1 Usage Restrictions

You may modify the Hoki Hoki source code however you please. You may not sell your modifications, and if you distribute modified binaries you must make the modified source available.

2 Requirements

To work with the Hoki Hoki source, you will need a C# compiler and the Managed DirectX SDK. I used Visual Studio .NET 2003 to create the game, and have included the solution, project, and resx files. I used the December 2005 SDK, but other versions may also work if the appropriate references are changed.

3 Overview

To be blunt, this isn't the prettiest source code you'll ever see. Hoki Hoki is one of the first projects I started with C# and Managed DirectX, and it's pretty hackish - no real organizational plan, lots of very unrefined code, inadequate documentation. The purpose of this document is to give you at least a general idea what's going on. I would not recommend this code as a learning tool to an inexperienced programmer, but I feel that, raw as it is, it may have some uses: There are a few interesting parts that could save others time working out an algorithm I've implemented here, and I would certainly like to see entertaining modifications to the game.

4 SpriteUtilities

SpriteUtilities is the set of 2D graphics classes I use in this game. It is the predecessor to another project of mine, Trans2D, and is based on the same idea of of using nested transformations to describe 2D objects. Here is an overview of its most important classes:

- `TransformedObject` is an abstract class describing a transformation. It has a number of properties like position, scale, rotation, and so on, as well as a list of objects that are inside it. The objects inside it inherit its transformations, in addition to its own: if object B is inside object A, A is at (10,0), and B is at (15,0), then on the screen B will appear at (25,0). If A is scaled 200% horizontally, B will appear at (35,0) on screen (since the 15 pixels between A's origin and B's have been doubled to 30), also horizontally stretched 200%.

- `SpriteObject` is derived from `TransformedObject`, and can draw a rectangular image using the transformations. There are other similar classes, like `SpriteTriFan`, `SpritePolygon`, and `SpriteText`, that are also derived from `TransformedObject` and similarly draw some image using the transformation.

- `SpriteTexture` defines a region on a `Texture`. The idea is to put many sprites in a single texture file, load that texture, then draw small pieces of it. The `SpriteObject` class handles the actual drawing.

5 Main loop

The main loop is in Game.cs, specifically in the `OnPaint` method. Every time `OnPaint` is called, the game checks how much time has passed and updates the simulation accordingly. The simulation uses fixed timesteps of length 0.0166, or 60 updates per second. The amount of time passed each frame is added to a variable `accumulator`, which is used to determine when it is time for an update. If 0.04 seconds have passed since the last update, the simulation will update twice and if 0.01 have passed it will not update at all. Once the simulation has been updated, the scene is drawn: all graphics are descendents of a SpriteObject called `root`, so drawing `root` draws everything. In this way, the framerate of the simulation is decoupled from that of the

display (which updates as often as it can), which helps keep the simulation stable while maintaining the smoothest possible display.

6 States

There are several GameStates (see GameState.cs), each of which signifies a certain mode the game is in - the menu screen, the actual game, and so on. When the game needs to change states, it calls the `setGameState` method, which performs the necessary cleanup for the old state and setup for the new. Cleanup involves nullifying all references to objects that are used exclusively for the current state. This is very important: if a reference to, say, a menu header graphic is not nullified, there will be a path to the texture it uses, so it will not be garbage collected and will thus stay in video memory. The texture for the interface is huge, so leaving it in memory when it is not needed could be problematic.

7 Maps

Maps files are divided into two sections, headers and objects. The headers, which all begin with a `#` symbol, are:

- `#NODECOUNT` [value]
- `#WALLCOUNT` [value]
- `#POLYCOUNT` [value]
- `#AUTHOR` [value]
- `#THEME` [lava.theme]

After the headers come a series of object listings. Lines beginning with a `>` symbol declare the type of object listed in the following lines, and the lines themselves are sets of comma-delimited values. The declarations and their corresponding line formats are:

- `>NODES` - x/8,y/8
- `>TRIANGLES` - node0,node1,node2

- `>LINES` - node0,node1
- `>PADS` - x/8,y/8,type
- `>SPRINGS` - x/8,y/8,rotation
- `>LAUNCHERS` - x,y,rotation,frequency,offset
- `>CATCHERS` - x,y,rotation
- `>SPRITES` - x/4,y/4,index,depth

Where nodes are give as values, they are actually stored as the index of the node as it appears in the list in the mapfile. Where x/8 and y/8 are given, the value is the coordinate divided by eight. For pads, the possible values for type are 0=start, 1=finish, 2=heal. Spring rotation is given as a number of 90 degree turns (so if it is 2, the spring is rotated 180 degrees). Similarly, launcher rotation is a number of 45 degree turns. Sprite index refers to the ordered list of textures in the theme file, and depth acts as a multiplier on the sprite's rate of movement relative to that of the level in order to make a parallax effect possible.

In the game, maps are built by the static method `Map.FromString`. It is mostly straightforward, with the exception of the code that joins walls together at nodes. The complexity of the problem is in drawing three or more walls without overlap so that the area inside them all can connect them smoothly. This is done by sorting the walls by angle relative to the node they meet at and finding where they intersect the walls adjacent to them. The walls are modified so that they share exactly one vertex with the adjacent wall where their edges intersected, then filling in the convex polygon defined by the set of intersections.

8 Collisions

Handling collisions between the helicopter and walls involves several intersection tests and some code to force the helicopter to remain inside the level. Each time the simulation is updated, the helicopter's endpoints move, both from its rotation and from the player's control. This motion may violate the notion that objects may not pass through each other, so the location of

the endpoints may have to be adjusted after the initial movement to conform to it. The simulation uses the technique Thomas Jakobsen described in "Advanced Character Physics" to do this. The code is rather inefficient, and I will admit that I decided it was "good enough" and moved on in order to finish the game quickly rather than improve it. An obvious improvement that could be made is the addition of a (working) bounding box test to cull the number of intersection tests down to a small few.

The endpoints are treated as particles with a current position and an old position, which given the simulation's fixed timestep implies velocity. Collisions with the walls are tested for in multiple ways:

- Helicopter through wall: consider the line segment connecting a helicopter vertex's current position with its position last frame. Even though it has technically traveled through an arc, the arc angle is so small that a straight line is an adequate approximation of the path the endpoint has taken. This assumption would of course break down without the guarantee of a frequent updates relative to the speed of helicopter rotation. If the segment intersects any walls, then the helicopter has collided. The endpoint is then projected out of the wall and a reacting force is applied to it.

- Wall through helicopter: it is possible for a wall corner to pass through the helicopter without the helicopter's endpoints crossing the wall. It is therefore necessary to test whether any wall vertices violate the helicopter. Since the helicopter is not stationary (it both rotates and, potentially, translates), there is not just a segment to test for intersection, but the area that segment has swept out over the past frame. This area ought to be an ellipse section, but I approximate it with a triangle. If a wall vertex is inside the triangle, both helicopter endpoints are projected away from the wall and a reacting force is applied proportionate to the where the wall vertex falls on the helicopter (if it is closer to endpoint A than endpoint B, endpoint A is knocked back farther).

- Mines: testing for mine collisions is pretty easy since they explode once touched - unlike the walls, we do not need to worry that the helicopter will be impossibly crossed through them when the next frame is drawn. To test whether the helicopter's line segment intersects the mine's circle, I find the line perpendicular to the helicopter that crosses

through the mine's center. I then find its intersection with the line that the helicopter's segment is on. If the intersection is not on the segment, I use the closest endpoint. The line segment connecting that intersection (or endpoint) with the center of the mine is the shortest possible. The helicopter intersects the mine if and only if that segment's length is less than the radius of the mine. If there is a collision, the mine explodes and an appropriate force is applied to each of the helicopter's endpoints.

After the endpoints are projected to satisfy the wall constraint, it is necessary to make sure they are still the right distance from each other since they represent a rigid body which cannot expand or contract. Adjusting them to satisfy this constraint may once again violate the walls, so both constraints are tested and fixed repeatedly, a process called relaxation. As naive as that approach may seem, it is remarkably stable - the helicopter is corrected further with each relaxation, and after only a few it is stable enough.

9 Recordkeeping

The game keeps records in two separate files: `scores` and `user`. In `scores`, there are two types of lines: a line beginning with `>` like `>e6b951a00c0106968e7821cc9d65620c` means that the following lines relate to the level with the given hash code. A line like `Max:9073:1:0` indicates that the user named Max completed the level in 9073ms, that the run was perfect (no walls hit), and that it was not done in easy mode. The latter two facts are given by the last two colon-delimited values 1 and 0, respectively. The game keeps track of the top three scores for every level played, and they are visible to all players. This allows a little competition for the top spot between people playing on the same machine.

In `user`, there are three types of lines: a line like #Max indicates that all of the following lines relate to the user named Max. A line like `>EASY:0` means that he does not have the game set to easy mode, which is the only user-specific preference that is stored. Finally, a line like `[hashcode]:2380:1:0` means that his best run through the level with the given hash code took 2380ms, was perfect, and was not in easy mode. A completed run in normal mode, no matter how poor the time, always overwrites a run in easy mode.

If the player has never completed a given level, it just doesn't appear in his section of the user file.

There are also a number of global options like controls and video settings stored in the `preferences` file. These are all indicated by a letter standing for the option followed by a colon and the value. The code for handling it is self-explanatory.

10 Setup

To create a setup like mine that will work on systems that do not already have Hoki Hoki installed, build the HokiSetup project and copy its output to ManagedSetup
Game. Zip the folder's entire contents up, as they are all necessary to install all the right DirectX stuff.

11 Contacting me

I am glad to provide personal help with the source code. I only ask that you do some research first where applicable. If you want clarification on how the wall-joining works, e-mail me. If you want help with file IO or rendering images, use Google. I get a lot of requests for help from people who do not have the basics of the language or graphics API down, which are things a person should know *before* trying to modify my malformed code :)

My e-mail address is max@flecko.net, and my AIM screen name is flecko14.