Final Report

Jakobsen Physics

Peter Giuntoli
4/16/2008

CONTENTS

OVERVIEW

The intent of the project was to create a physics engine that would be used in the creation of an American Football simulation. This idea was to model a system after the popular Euphoria[1] engine. This meant that animation of characters should be able to work seamlessly with the physics engine, creating lifelike animations in a variety of physics based situations (primarily tackling).

The initial plan was to create an engine based off of the paper Advanced Character Physics[2] by Thomas Jakobsen. This system resorts to handling physics by placing particles on objects which act as physical points of interest. These particles are then connected by constraints that attempt to keep particles at a specified distance from each other. If enough constraints are placed, this simple system of particles and constraints can model a rigid body effectively, as well as allow for simple creation of ragdolls.

ENGINE

Being the author's first attempt at a game engine, there were some snags in the process but the engine ended up being passable for the task at hand. If the engine were transitioned to be used in a game, major overhauls would most likely be necessary due to ease of use and performance issues.

GRAPHICS

The graphics engine consisted of a basic DirectX framework which was constructed with major help from a few key websites[3]. Functionality includes the loading of .x models and object basic lighting. Models can be drawn in wireframe or filled mode depending on user preference. There are also basic debugging functions which allow the user to draw wireframe boxes and lines anywhere in the world. This feature was invaluable when testing points of collision.

INPUT

Input is handled easily by using the Simple DirectMedia Layer[4]. The library enables the easy creation of Win32 windows as well as a messaging system that wraps around the usual Win32 winproc. The messaging system can be used to catch windows messages and also gives the user input polling functionality if it is desired. Without having to deal with the normal Win32 frustrations, the process of building an engine was made much easier.

[1] http://www.naturalmotion.com/euphoria.htm
[2] http://www.teknikus.dk/tj/gdc2001.htm
[3] http://www.toymaker.info/Games/html/direct3d_faq.html
[4] http://www.libsdl.org/

Some complications that arose were the fact that a user can hold down on a key and briefly move the mouse outside of the primary window. This can be done without the user knowing (the mouse cursor is hidden and always brought back to the middle of the screen each loop) but will cause the screen to stop receiving windows messages. If the user lets go of a key while the cursor is outside of the window it will still be registered as being down. Because of this, there are points in time where the user's camera may continue to move in a certain direction while the user has released all keys. Not much effort was put in to solving the problem, as it occurs very rarely, but an easy fix would be to use the normal Win32 winproc which would more than likely be done on a larger scale project.

MEMORY MANAGEMENT

In hindsight, memory management was probably unnecessary but was put in so that the engine would feel more complete. The memory manager uses boost::pools[5] to do most of the heavy lifting. The memory manager handles creation of all objects in the game except for the engine itself. This helps to ensure that there are no memory leaks in the program.

FILE IO

To make the importing of levels and models easier, TinyXML[6] was utilized. This XML file loading system made the process of creating rigid bodies and ragdolls multitudes easier than having to code all by hand.

PHYSICS

The most important part of the entire engine only began to come together in a workable state during the final stages of the project. The physics engine uses Jakobsen constraints, as discussed earlier, to model rigid bodies and ragdolls. A drawback comes from the fact that only rectangular prisms are able to be used with the Separating-Axis Test, so the ragdoll is made completely of rectangular objects. This limitation does not entirely hinder the ragdoll model, but if the system was brought in to a game the limitation would most likely need to be removed.

In essence, the physics loop consists of four steps. First, all particles have external forces applied to them. In the simulation, this only consists of gravity.

Next, all particles have their positions updated. This process is done by using a Verlet integrator. At this point in the simulation, wind resistance is applied to the particles. This wind resistance ends up being the only resisting force. The force is applied by slowing down the velocity of the particles by a scalar. In effect, this will bring most objects to a stop but has some unwanted results (Note: ragdolls still move along the ground because of the constraint solvers). The major

[5] http://www.boost.org/doc/libs/1_35_0/libs/pool/doc/index.html
[6] http://www.grinninglizard.com/tinyxml/

problem is that terminal velocity, when using a normal gravitational acceleration, appears very low. This can be combated by increasing gravity until the “look” is right but the speed in which ragdolls slide along the ground also increases.

Broad phase collision detection is then performed by doing sphere-sphere and AABB-AABB checks. If objects are not determined to be separated after the broad phase step, a separating axis check is performed. This test is optimized for rectangular prisms which means that only 15 possible separating axis need to be checked.

If objects are determined to be interpenetrating, the axis of minimum penetration is chosen to be the collision normal. Penetration depth is found by taking the dot product of the collision normal with the maximal penetrating vertex of one object subtracted by the dot product of the collision normal with the minimal penetrating vertex of the other.

During collision resolution, the type of collision is determined by using the collision normal and dotting all vertices of an object against it. Any vertices that fall within an epsilon (currently .01) of the maximal or minimal penetration (depending on the object) are saved. From this information, it can be determined if there is a Face-Face, Face-Edge, Face-Vertex, Edge-Edge, Edge-Vertex, or Vertex-Vertex collision. All collision resolution types end up moving a point on all collision features by a magnitude equal to half of the penetration depth in the direction of the collision normal. Finding the collision point, and the magnitude each vertex on the collision feature must move, is simple for some cases but proved difficult for others, most notably Face-Face collisions.

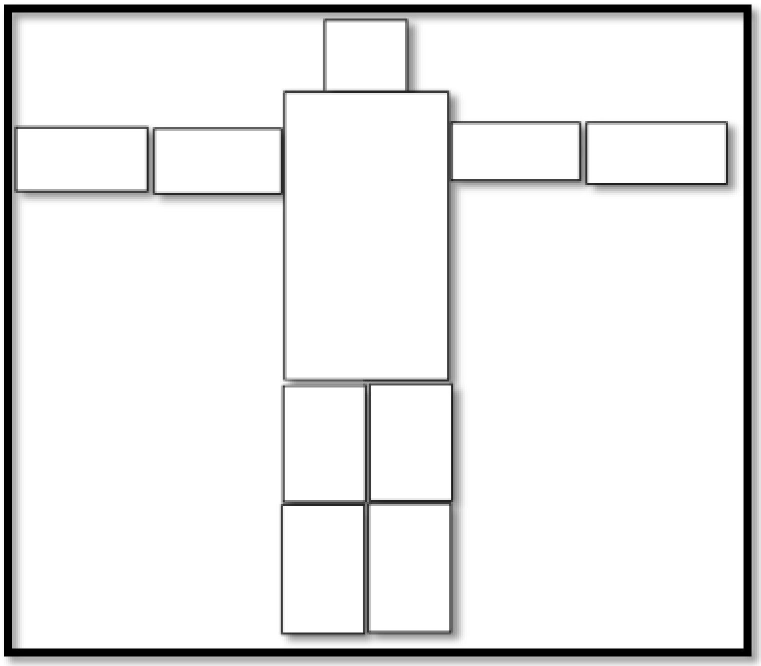
During Face-Face collisions, if a face is completely contained within the other, the barycenter of the contained face is used as the collision point. If neither face is contained within the other, both faces are clipped. The resulting shape’s barycenter is then used as the point of collision.

To move a face after determining the point of collision, a 4x4 matrix is constructed which contains all points of the face. This can be done because a face will always consist of four or less points due to objects being rectangular prisms. The inverse of the matrix is then multiplied against the point of collision to find a set of scalars that will be used to move the vertices of the face. If an inverse cannot be found, a guess at those scalars is made by iterating the function: $W_{k+1} = (I - gP)W_k + gX$. Where X is the collision point, I is the identity matrix, P is the 4x4 matrix of vertices, and g is some small scalar (.01 in practice). W is then used as the set of scalars to move the objects.

After collision resolution, constraints are satisfied by iterating three times over all constraints in an attempt to keep particles a set distance from each other. The only constraints implemented were stick constraints. A valuable addition might be to include softer constraints that would allow objects to be within a certain range from each other, in effect, having a minimum and maximum distance that they could lie. It is believed that this addition could create a more believable ragdoll.

RAGDOLL SETUP

The ragdoll is set up by using ten boxes and twenty constraints connecting them. This makes a total of 80 particles and 200 constraints per ragdoll.



Each limb is connected by using two stick constraints that act as an "elbow". Setting up constraints in this fashion allows limbs to move freely along a single plane. The only connected piece that uses more than two constraints is the head, which is connected by four constraints.

The setup of the ragdoll causes self collisions almost every frame. Although with small numbers of ragdolls this does not create a significant slowdown, once large numbers are simulated this can create a bottleneck.

Some possible speedups would be to create a bounding hierarchy for the ragdolls. Instead, each box maintained its own bounding sphere and axis aligned bounding box.

WHAT WENT WRONG

The Jakobsen physics engine seemed to hinder project development more than help it. The majority of the time spent on the project was spent on getting rigid bodies working. There were many iterations on collision detection and resolution until finally settling on SAT and letting the Verlet integrator handle resolution.

Originally, it was planned to do collision detection between faces and particles as well as constraints and constraints. Although particle face collision detection was easy enough, constraint vs constraint detection ended up taking a lot of time to figure out a solution for, and in fact, a suitable solution was never found. The final attempt at a solution involved finding the closest points on the two constraints at the beginning and end of the frame. Vectors were formed between these points and if the dot product was negative, a collision was determined to have occurred. Besides the fact that for a single box vs box test this required 324 tests (18 constraints per box) causing a significant slowdown, the collisions detected were not guaranteed to be collisions.

The second biggest problem came when trying to find an affine combination for a point on the face of a box. The affine combination is required for collision resolution since I do not use impulse or inertia tensors. A simple matrix inversion can sometimes solve the problem, but the majority of the time this is not the case. In a case where a matrix cannot be inverted, a linear equation is iterated over until a guess at the solution is found. In some cases, the guess is so far off that the simulation breaks and objects fly off to infinity.

WHAT WENT RIGHT

The finished engine has impressed a couple other DigiPen students who are interested in using some of the functionality towards a senior game. It is yet undetermined if the ideas will carry forward in to a game project but the respect of his peers is something the author greatly appreciates.

Having ragdolls implemented in the last few weeks of the class was also very satisfying as there were pressing concerns that the rigid body portion of the simulation might not be serviceable. All fears were alleviated when the separating axis theorem was implemented and collision resolution became believable. Unfortunately, this came near the end of the semester so not all of the original goals of the project were able to be completed.

PROJECT FEATURES/CONTROLS

Name	Summary	Key Command
Scene 1 – Ragdoll	This scene consists of a single ragdoll in the center of the world	1
Scene 2 – Box Pyramid	This scene consists of a pyramid of boxes that vary in size.	2
Scene 3 – Box of Boxes	This scene consists of 3, 3x3 layers of same size boxes.	3
Clear Scene	This will clear the entire scene of objects.	0
Change Draw Options	Switch between three draw modes. **Filled** shows the objects in a normal fashion. **Wireframe** will display the wireframe version of objects. **Constraint** will show all constraints in the world as red.	Tab
Draw Collision Vertices	Draws green boxes around all vertices that are being updated do to object collisions. Note, these do not include vertices update by ground collision.	C
Move Camera	Will move the camera around the world in a normal first person camera fashion.	W,A,S,D
Adjust Camera Orientation	Adjust the focus of the camera by moving the mouse around.	Mouse Movement
Change Gravity	Cycle the gravity between two modes. **Normal** will have gravity always point downwards. **Point** will make all gravity located towards a single point above the ground plane.	G
Change Force of Gravity	Change how much force is applied to objects due to gravity.	Y, H
Change Wind Resistance	Change how much "wind resistance" is in the world. This is the only dampening force.	U, J
Pause	Pause the Simulation	Spacebar