|  |
| --- |
| **Concordia University**  **Department of Computer Science**  **and Software Engineering** |

|  |
| --- |
| Animation |

|  |
| --- |
| **Final Project – Jenga Simulator** |

|  |
| --- |
| **COMP 477**  **Winter 2013**  **Professor: Dr. Grogono** |
| **By: Patrick Modafferi 9401377  Christopher Di Fulvio 9614605**  **04/22/2013** |

We certify that this submission is my original work and meets the Faculty’s Expectations of originality

# Instructions

## Starting the game

1) Go to the JengaSimulator folder

2) Open JengaSimulator.sln with visual studio 2010

3) Press F5 to compile and run in debug mode

If you do not have visual studio

(Should work on all school computers and computers with .NET installed)

1) Go to JengaSimulator \ JengaSimulator \bin\x86\Debug

2) Double click on JengaSimulator.exe

Code can also be found on GitHub:

<https://github.com/Polyphechris/JengaSimulator>

## Controls

|  |  |
| --- | --- |
| TOGGLE KEY(S) | ACTIONS |
| F11 | Full Screen Mode (Recommended) |
| SPACE | Pause / Navigate Menus |
| Space , I | Instructions screen showing all available controls |
| 1 / Shift + 1 | Toggle Rain |
| 2 / Shift + 2 | Toggle Snow |
| F / Shift + F | Toggle Fireworks |
| G / Shift + G | Toggle day times |
| B / Shift + B | Hide/Show bounding boxes |
| 3 / Shift + 3 | Toggle rotational physics on/off |
| CAMERA CONTROLS |  |
| LEFT/RIGHT ARROW | Pan Left and Right |
| UP/DOWN ARROW | Pan Up and Down |
| I/K | Rotate Camera Up and Down |
| J/L | Rotate Camera Left and Right |
| Q/E | Move Camera Forward and Backward |
| Z/C | Rotate the world |
| X / Shift + X | Reset camera with many or one block |
| HAND CONTROLS |  |
| W/S | Move Hand Up and Down |
| A/D | Move Hand Left and Right |
| Left Click (Hold) | Poke hand forward |
| Right Click + W/S | Rotate Hand Up and Down |
| Right Click + A/D | Rotate Hand Left and Right |
| \*Shift means Left Shift |  |

# C:\Users\Patrick\Documents\GitHub\JengaSimulator\ClassDiagram.pngArchitecture

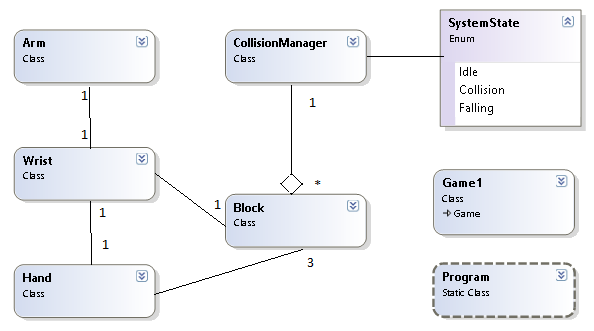


Figure 2‑ Class Diagram for Particle Systems

Figure ‑ Class Diagram for Game Elements

# Particle Systems

For the assignments we created a particle system of particle systems. To accomplish this, all particle systems contain an object called “Bursts” which is a list of Particle systems. This creates a recursive type of relationship in the update and draw functions used for fireworks and shooting stars.

## Star System

The static stars obtain a random 3D coordinate based on two angles (Phi and Theta) mapped to a large sphere. The complexity rises when shooting stars are added. Each shooting star is more than just another particle; it also has a trail and behaviour of its own. Therefore, when we reach the specified spawn rate, another shooting star is added to the “Bursts” discussed earlier. Their initial velocity is then calculated as a tangent to the sphere located at a random static star’s location.

## Fireworks

The fireworks are made of three main components: Firework Rockets, Firework Explosions and Firework Rocket Trails. Each of these components is considered as its own particle system. The rockets get fired upwards with a large vertical velocity and random sideways direction. When the vertical speed reaches zero, the rocket explodes and a new system is created at the source of the rocket’s explosion. Random lifespan and color are given along with directions calculated like in the star system. When all the exploding particles expire, we re-spawn the rocket at the initial source with a new random direction. In addition to what was done for the assignment requirements, extra look and feel was expanded on to make the project more interesting. To avoid recurring patterns and add realism, there is a random offset in the x and z directions for the fire

## Trails

Fireworks and shooting stars contain trails. These trails are simply particle systems that get new particles added to them every time the object updates. Basically, a new stationary particle gets created at the location of the leading object. These trailing particles then have their alpha value decreased and die when the alpha reaches zero.

## Weather System

For this project, we thought it would be interesting to expand on the particle systems made for the assignments and introduce a weather system. The user can toggle between rain, snow and fair weather. Each state modifies the sky color and ground material properties. The snow introduces a simple matte effect with white. The rain adds a specular color with a high shiny constant.

### Rain

The rain particle system simply requires a series of raindrops with downwards velocities. These raindrops start from a random height then drop with an acceleration equal to gravity. Their y-direction scale depends on their downwards velocity to give the appearance of the long faster drops elongating and stretching out.

### Snow

The snow draws a lot of algorithms from the rain system. The main difference is in the aesthetics on the particle itself. To add more realism, a “flurry-ness” value is introduced. Each particle falls in the general downwards direction, but obtains a random value for X, Y and Z (between negative FLURRY and FLURRY) each frame to offset its position. This adds the effect of being in a blizzard.

# The Arm

The hand is composed of an arm, which contains a wrist, which contains a hand. We tried as much as possible to follow the guidelines taught in this course. For articulations we keep a hierarchy of body parts that are interconnected. Each connection represents some degree of freedom around which the arm can move. Let us explore in more detail the separate articulations where motion is permitted.

* **Wrist**

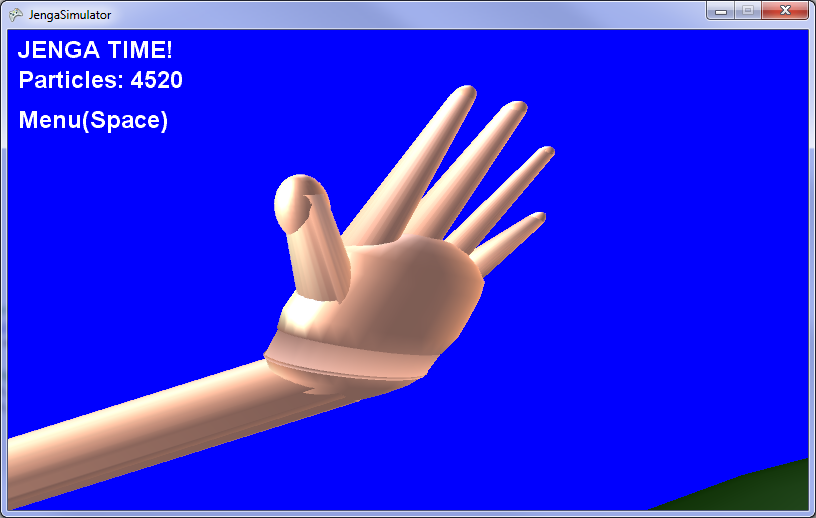
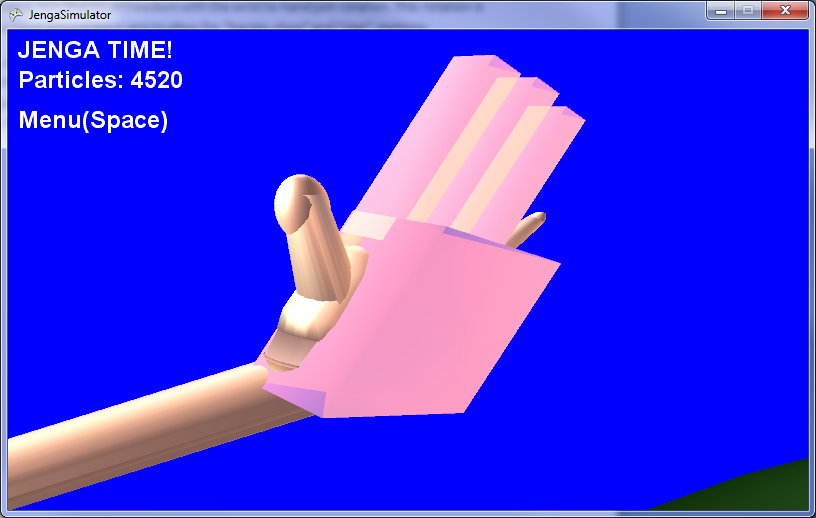
The wrist is positioned relatively to the arm and rotations on the arm get applied to the wrist. The arm is constrained to move linearly within an imaginary 3D box. All these translations get applied to the wrist directly since it is the first visual component to the arm.

* **Hand**

The wrist contains a hand which is also positioned relatively to the wrist and move linearly with it. When the wrist receives a rotation, that rotation also gets passed along to the hand object since they are locked on the z-axis articulation and that is the only permitted rotational force for the wrist. The hand also introduces a new degree of freedom with the wrist to hand join rotation. This rotation is applied only to the hand around the x-axis to allow for “karate-chop” and “slap” motions.

For collision, a series of bounding boxes which act exactly like the blocks, are set to cover the surfaces of the hand. To be specific, there are 3 long boxes for the 3 larger fingers in the middle of the hand and one larger box for the palm. We opted to use only the fingers for poking so that our hand could actually push and pull blocks in a dragging motion like a real hand.

4‑1 The Hand in action and The Hand with the collision boxes



# Physics and Collision

The physics of the system is very straightforward. At all times, the blocks are affected by a -10 unit force with a downwards direction, acting as gravity. The block's position and velocity are calculated using simplified kinematics formulas:

*p1 = p0 + v \* t and v1 = v0 + a \* t.*

The system is either in a state of "idle" or "collision". The system is automatically set to a state of collision when any collision happens between the hand and any block. The system resets back to the idle state when the velocity of each block is within a certain threshold (they are moving slow enough to say that they are no longer moving). When in an idle state, the update method of the blocks are no longer called so that they do not move with their tiny velocity.

When a block collides with another, they create impulses against each other. These impulses are calculated based on which side of the block the colliding block is penetrating it (ex. if one block collides with another from the top, it is penetrating in the -Y direction, giving itself a positive Y impulse, whereas the block on the bottom receives a negative Y impulse). Angular velocity is calculated using the cross product of the total acceleration of the block with vector representing the direction from the center of the block with the center of the colliding block (See figure below for a rough sketch).

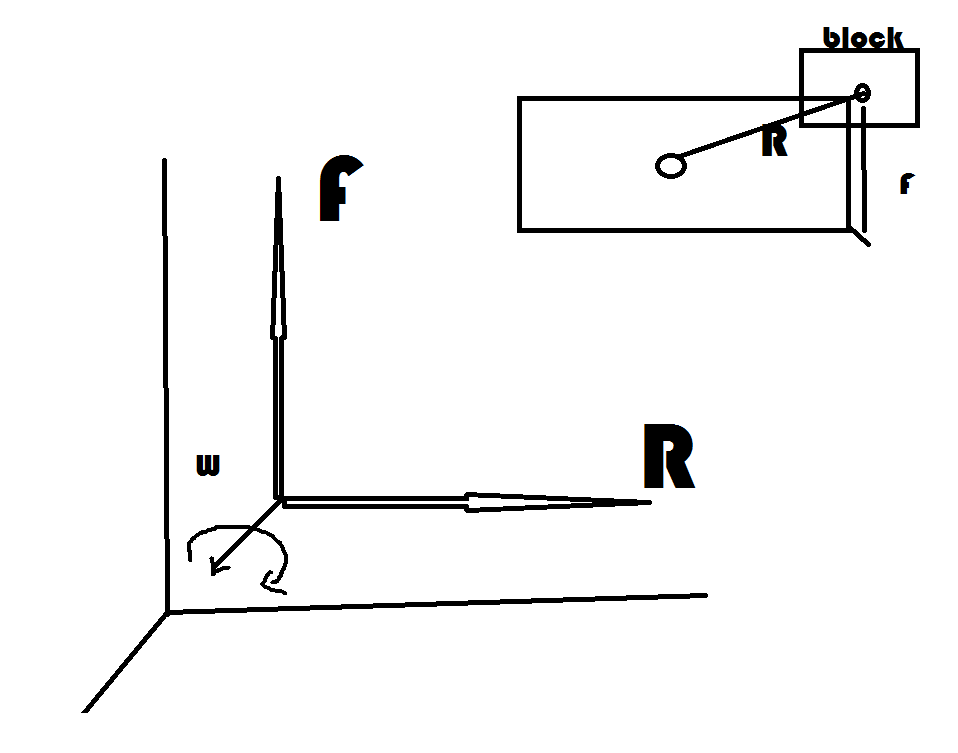


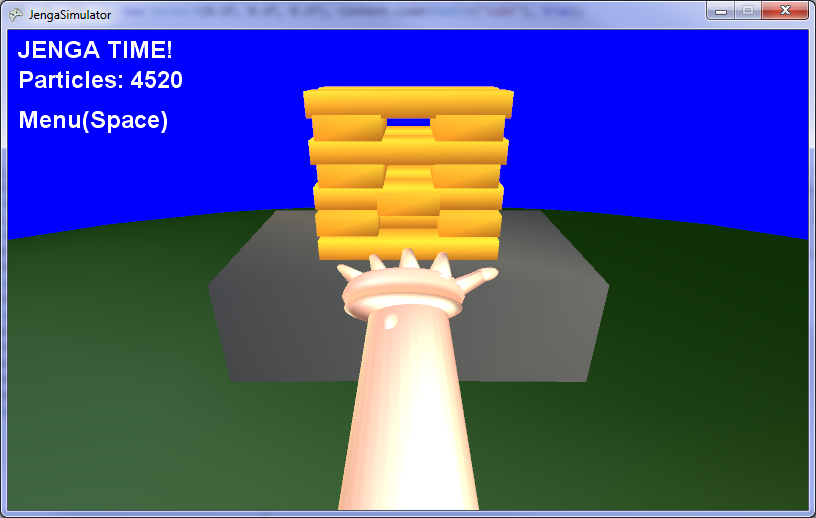
Figure 5‑1 Rotational Force Calculation

The resulting formula for the rotational can be simplified to:

*I = R X F*

Where R and F represent a 3D Vector and I acts as a rotational acceleration in 3D space acting on Yaw, Pitch and Roll.

In order for blocks to be able to rest upon each other, each block has an extra extended bounding box that protrudes in the negative Y direction. This bounding box checks to see if there is a block underneath it, but not colliding. If this is true, then the current block is set to "resting", which gives the block an upwards force equal to gravity so that it no longer falls. Once a block's protruding bounding box no longer collides with anything, the block is no longer resting and resumes falling with gravity. Certain blocks are set to "static", so that these static blocks are not influenced by gravity (these two blocks are the platform and the "earth").

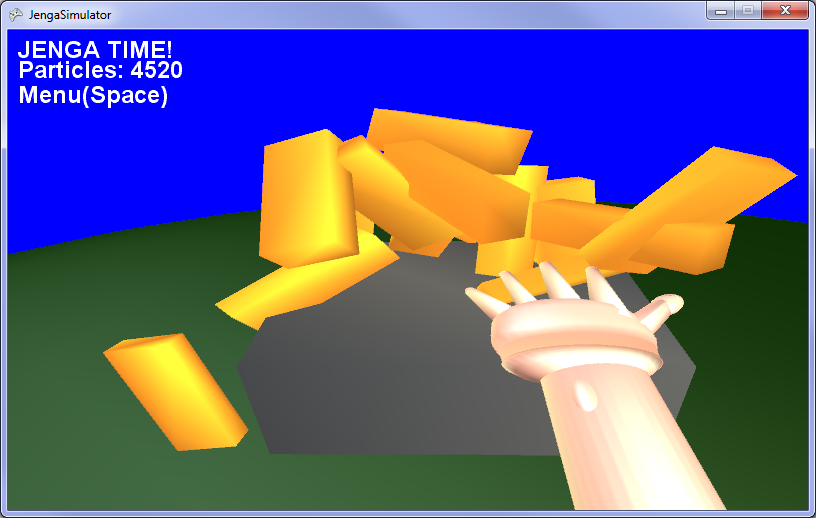


5‑2 Blocks falling into their resting state

# Results

The biggest problems we faced while working on the project was the collision and physics of a multi-block system. Mostly, the collisions between the hand and the blocks were giving a lot of bugs with random and numerous impulses being created at once, and sometimes with impulses being too high because of accumulation of acceleration from blocks not resolving their collision correctly.

One of the most challenging parts of the project was getting the blocks to fall lifelike. This involves having very good collision resolution, as well as realistic impulses, friction and angular acceleration. Unfortunately, as so much time was focused on getting the collisions and collision resolution fully functional there was not too much time to get the angular acceleration working the way we wanted it to be. There is a beta version of the angular component working in the software, but the main issue is that XNA creates bounding boxes for models that are permanently axis aligned. If a model is turned, it just creates a the smallest axis aligned bounding box that can possibly fit in the model, even if an object aligned bounding box is what is required. The below image shows that even though the rotational collision detection is acting strange, the system still allows blocks to be genuinely resting, aka there are no blocks floating in the air:



What makes this project interesting and relevant to the course is that it covers some key aspects of animation, such as collision resolution, human body simulation and particle systems. It covers the fundamental parts of creating a world in which its objects can interact with each other, which is very important when creating an interactive video game. For the future, the program could become a fully fledged Jenga simulator, where the rotational physics are working properly, the collision is 100% functional, and the player is able to pinch their fingers as to remove blocks from the tower as you would when playing the actual game.

Figure 6‑1 Rotational Resting

# Credits

|  |  |  |
| --- | --- | --- |
| **Patrick Modafferi** | **Chris Di Fulvio** | **Teamwork** |
| * Particles systems (Snow, Rain, Fireworks, Stars) | * Linear Collision Resolution | * Physics (Gravity, Impulses, Friction) |
| * Hand motion programing | * Hand modeling (art asset) | * Hand Collisions * Rotational Physics |
| * Camera and Menu system | * Report section 5-6 |  |
| * Report section 1-2-3-4-7 |  |  |