Animator

April 3, 2018

Objectives

- Modeling the Scene
- UI
- Texture Mapping
- Keyframe Interpolation
- Sound (synchronized to collisions)
- Static Collision Detection
- Dynamic Collision Detection
- Physics Engine (gravity)
- Reflection Mapping
- Cel Shading

Purpose

The purpose of this project is to develop an application that allows users to create animations by specifying keyframes. The models will behave according to some laws of physics, such as gravity and the conservation of momentum.

Goals

Interpolating between keyframes

- Animating transformations (translation, rotation, scaling) of a model from one keyframe to the next.

Behavior of the model after the last keyframe

- Animating the effect of gravity on the model after the last keyframe

Collision Detection

- Getting the collision time between two models and the normal of the face it collided with

3D Elastic Collisions

- Having the models bounce back after collision, following the conservation of momentum

Sound

- Play the collision sound when two models collide. The volume is based distance from the camera and speed of the model going into the collision

Texture Mapping

- Save the UV coordinates of each model that gets imported and make changes to the Fragment Shader to get the color of the texture at the current coordinate

Skybox

- Add a skybox to the scene, which is a cube around the camera that is only transformed by camera rotation, not translation

Reflection Mapping

- Make changes to the Vertex Shader to obtain the reflected ray, and the Fragment Shader to get the color of the skybox at the point where the reflected ray hits

Cel Shading

- Make changes to the Fragment Shader to output the interval that the color belongs to

Statement

Traditionally, to create an animation, a master artist draws the keyframes while sweatshop artists draw each frame between the keyframes through inbetweening. This application allows users to create animations much easily and quickly. The user only has to set keyframes by transforming 3D models in the scene and the program will interpolate between keyframes for them.

Similar animation software already exists, such as Blender and Synfig (2D only). However this application not only interpolates linearly between keyframes, it also simulates some properties of physics. For example, to animate a bouncing ball, the animator only has to set gravity for a model and press play. The ball will fall by itself without additional keyframes, and every time it hits the ground, it will bounce back according to elastic collisions. In existing animation software, users will have to specify keyframes for each bounce, as opposed to only needing the initial keyframe in this application.

With this application, creating an animation would be as simple as moving objects around in real life. The keyframe interpolation will move models from one position to another, while the physics engine causes the models drop and bounce according to laws of physics.

Webpage: http://acmyu.github.io/animator.html

Manual

Compiling

Compile the program on the Linux command line in the project's directory with:

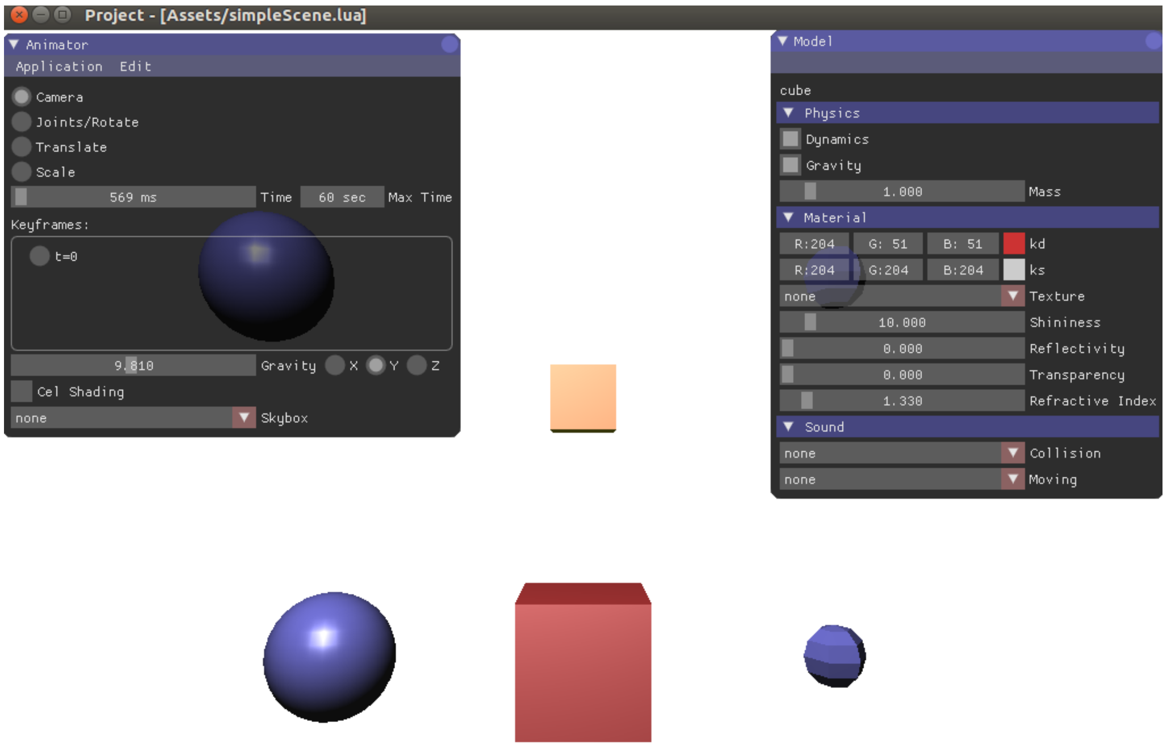
```
export ORIGIN=$(pwd)
premake4 gmake
make
```

Starting the program

Run the program on the command line with:

```
./Project <.lua or .ani file>
```

A .lua file opens a scene that is not yet animated. For example, `Assets/simpleScene.lua` opens to the scene below:



Camera
Joints/Rotate
Translate
Scale
569 ms Time
60 sec Max Time
Keyframes:
t=0
9.810 Gravity X Y Z
Cel Shading
none Skybox

cube
Dynamics
Gravity
1.000 Mass
R:204 G: 51 B: 51 kd
R:204 G:204 B:204 ks
none Texture
10.000 Shininess
0.000 Reflectivity
0.000 Transparency
1.330 Refractive Index
none Collision
none Moving

The Animator Interface on the left will only have the initial keyframe, t=0. To animate it, the user will need to add more keyframes.

A .ani file opens to an animation that was previously exported. The Animator Interface will have keyframes that were previously saved. The user can play or modify the animation. To export the new changes overwriting the .ani file, press x on the keyboard.

To play an animation, press the SPACE bar.

Transforming the Camera and Model

The user can transform the camera or the selected model after selecting the mode on the Animator Interface on the left. For model transformations, select a model by left clicking on it. A selected model will tint yellow. Select multiple models by holding down the CTRL key. The modes are:

- **Camera mode:** Clicking and dragging the mouse on the screen will rotate or translate the camera
    - **Left click:** translate camera on the x- or y-axis. The amount translated on the x-axis depends on the change in the x position of the mouse, and the y-axis depends on the change in y position
    - **Middle click:** rotate camera on the z-axis. The amount rotated depends on the change in the y position of the mouse
    - **Right click:** rotate camera on the x- or y-axis. The amount rotated on the y-axis depends on the change in the x position of the mouse, and the y-axis depends on the change in x position
    - **Scroll:** translate camera on the z-axis
- **Joints/Rotate mode:** Clicking and dragging the mouse on the screen will rotate the selected models. Rotating a joint is currently untested and may not work in an animation.
    - **Middle click:** rotates the model on the y-axis, depending on the change in the y position of the mouse
    - **Right click:** rotates the model on the x-axis, depending on the change in the y position of the mouse
- **Translate mode:** Clicking and dragging the mouse on the screen will translate the selected models.
    - **Middle click:** translates the model on the z-axis, depending on the change in the y position of the mouse
    - **Right click:** translates the model on the x- or y-axis. The amount translated on the x-axis depends on the change in the x position of the mouse, and the y-axis depends on the change in y position
- **Scale mode:** Clicking and dragging the mouse on the screen will scale the selected models.
    - **Middle click:** scales the model on the z-axis, depending on the change in the y position of the mouse
    - **Right click:** scales the model on the x- or y-axis. The amount scaled on the x-axis depends on the change in the x position of the mouse, and the y-axis depends on the change in y position

The Animator Interface

The Animator Interface is the window on the left that lets the user change the mode, change the current time, select keyframes, and set properties that apply to the entire scene.

The user can minimize and reopen the windows by double clicking on the top bar of the window.

To change the current time, drag the **Time** slider. This will move the animation forward in time similar to a playback bar. The models and camera will move based the time that the user dragged the slider to. The time displayed on the slider is in milliseconds.

The **Max Time** slider specifies the length of the animation. The user can lengthen or shorten the animation by dragging it. The default is 60 seconds, the minimum is 1 second, and the maximum is 300000 seconds. Shortening the animation will delete all keyframes past the max time.

The list of **Keyframes** allows the user to jump to existing keyframes and edit them. The user can select the keyframe by clicking the radio button corresponding to the time of the keyframe, which will move the scene to its state at that keyframe. The user can delete any keyframe except for the initial t=0 keyframe by clicking the Delete button right beside it.

The **Gravity** slider allows the user to change the acceleration of gravity. The default value is 9.81 m/s^2 where each pixel corresponds to a meter. The **X**, **Y** and **Z** radio buttons let users specify the down axis, which is the axis along which the models fall.

The **Cel Shading** checkbox allows users to turn cel shading on or off.

The **Skybox** dropdown allows users to select a skybox to apply to the scene.

The Model Interface

The Model Interface allows users to edit the last selected model. This window only appears when a model has been selected. The interface has the following sections:

- **Physics**: This specifies which laws of physics apply to the model
  - **Dynamics**: If checked, the model will move between keyframes according to kinematic equations (instead of linear interpolation), bounce off in a collision (instead of stopping when it collides) and continue moving in constant motion after the last keyframe (instead of stopping).
  - **Gravity**: If checked, the model will drop down and move according to projectile motion after the last keyframe based on the acceleration of gravity specified in the Animator Interface. If not checked, the model will move in constant motion or stop after the last keyframe depending on the Dynamics checkbox.
  - **Mass**: This affects the behavior during a collision. According to the conservation of momentum, the velocity after collision depends on the mass of the model and what it collided
- **Material**: This lets users change the appearance of the model
  - **kd**: This color editor lets users change the diffuse color of the model by dragging each RGB value
  - **ks**: This color editor lets users change the specular color of the model by dragging each RGB value

    - **Texture**: The user can select a texture from the dropdown that will be applied to the model through texture mapping
    - **Tint**: This slider appears after a texture has been selected. It tints the texture based on the color in kd.
    - **Shininess**: The strength of the specular light. To turn off specular lighting, drag it to the maximum value.
    - **Reflectivity**: How much the model reflects the skybox. It will be completely reflective if dragged to the maximum value
    - **Transparency**: How much the skybox can be seen through the model. It will be completely transparent if dragged to the maximum value
    - **Refractive index**: Specifies the refractive index of the model if it has transparency. Air has refractive index of 1, water is 1.33
- **Sound**: Lets users specify the sounds the model will make when collided or moving.
    - **Collision**: This sound will play once when the model collides with another model. The volume depends on the speed before collision and the distance to the camera.
    - **Moving**: This sound will play in a loop as long as the model has non-zero velocity. The volume depends on the speed it's moving and the distance to the camera.

Keys

Additional features can be used by pressing certain keys on the keyboard. These are:

- u: Undo the transformation to the model or camera. Each keyframe has its own undo stack
- r: Redo the transformation
- x: Export the animation to Animations/new.ani. This saves all keyframes and settings, so the user can import it with `./Project <path to .ani file>`
- q: Quit the program

Creating a Simple Animation

1. Create a .lua file with models you will use. The models must be the existing .obj files in the Assets folder.
2. On the command line, run

```
./Project <path to .lua file>
```

3. Select a model on the screen. Click on the Model Interface > Physics > Gravity checkbox. Press the SPACE bar. This is a simple animation where the model drops down based on gravity. Uncheck the Gravity checkbox.
4. Drag the Animator Interface > Time slider forward in time. Select a mode on the Animator Interface. Do a camera or model transformation. A new keyframe should

appear on the list of keyframes in the Animator Interface. Select the t=0 keyframe. Press SPACE bar. The model or camera should move based on keyframe interpolation.

5. Press x on the keyboard to export the animation. It will get exported to Animations/new.ani. Rename this file.
6. Quit the program and run

   ```
   ./Project <path to .ani file>
   ```

   This imports your animation to play or modify

Examples

```
./Project Animations/spirited-away.ani
```

- An animation based on a scene from the anime Spirited Away. Press SPACEBAR for animation to start

```
./Project Animations/orbs.ani
```

- An animation with 6 spheres that have reflectivity or transparency. Press SPACEBAR for animation to start

```
./Project Animations/stairs.ani
```

- An animation where the cube will jump down the stairs. Press SPACEBAR for animation to start

```
./Project Assets/simpleScene.lua
```

- A scene with 6 models. Select a time for the new keyframe by dragging the Time bar on the left ui window. Select a model by clicking on it. You can add physics, texture, sound, etc. to the model on the right ui window. Select Rotate, Translate or Scale mode on the left ui window and transform the model. Then press SPACEBAR for animation to start.

Organization

Directories

- All source files (.cpp and .hpp files) are located in the main project directory
- **Assets/**: Contains the .obj files, Vertex Shaders, Fragment Shaders and example .lua files. Has subdirectories:
  - **texture**/: Textures (.bmp files) that can be selected for texture mapping
  - **sound**/: Sounds (.wav files) that can be used for a model's Collision or Moving sounds
  - **skybox**/: Folders containing skyboxes. Each folder contains front.bmp, back.bmp, left.bmp, right.bmp, top.bmp, bottom.bmp, which are used as textures for each side of the skybox
- **Animations/**: Contains .ani files which are exported by the program to save keyframes and settings of animations. They can be imported with `./Project <path to .ani file>`
- **lib/**: contains libsfml-audio.so and libsfml-system.so, which are libraries used for playing sound
- **include/SFML/**: contains SFML header files. Only SFML/Audio.hpp is included by Project.cpp for playing sound.

Source files

- **Project.cpp, hpp**
  - Imports the .obj meshes, lua file or ani file
  - Initializes values
  - UI
  - Handles user input, such as mouse input
  - Draws each frame, updates shader uniforms
  - Saves keyframes and animation settings
  - Interpolates between keyframes
  - Handles collisions (calculates velocity after collision, plays collision sound)
  - Saves and applies model states
- **SceneNode.cpp, hpp**
  - Does model transformations (translate, rotate, scale)
  - Saves model settings (sounds, materials, physics)
- **GeometryNode.cpp, hpp**
  - Parses .obj files into meshes, consisting of vectors of vertices, normals, uv mappings, and faces
  - Applies transformations to meshes
  - Calculates the bounding sphere of this model
  - Collision Detection algorithm

Implementation

Model Transformations

Each model has a NodeState containing the coordinates of the “center”, angle of rotation, amount of scaling, etc.

The center of the model is initially the origin. When a model gets translated or rotated, the transformations are also applied to this point. This would keep track of the model’s current location in the world coordinate system. To translate to a new location specified by *center*, it would translate by the amount (*center – oldCenter*). More generally, to transform to a new value, the amount it needs to transform is (newValue - oldValue)

Rotating the model will rotate it about its own x, y or z-axis. Each time the model gets transformed, its x, y and z-axis get transformed too. This would make it easier to save the angle of rotation for each axis, since the order of the transformations wouldn’t matter. If it was rotated about the world coordinate system’s axis, the order of transformations would matter. For example, rotating it first and then translating would be different from the other way around.

Linear Interpolation between Keyframes

Each keyframe is saved as an instance of the Frame class containing the NodeState for each model, the time of the keyframe, the camera transformations at this keyframe, an undo stack, and the initial velocity, displacement and acceleration to get from this keyframe to the next.

For linear interpolation, each time the program draws a frame, it interpolates the transformations for the current time t between the previous and next keyframes [1].

$$t = \frac{currentTime\ -\ prevKeyframe.time}{nextKeyframe.time -\ prevKeyframe.time}$$

Then for each model:

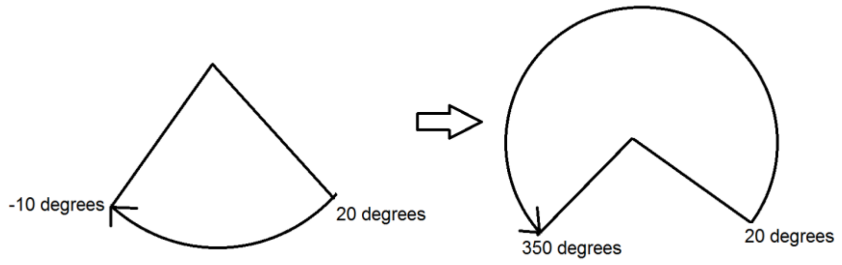
$$center = prevKeyframe.center + t * (nextKeyframe.center - prevKeyframe.center)$$

$$angleX,Y,Z = prevKeyframe.angleX,Y,Z + t * (nextKeyframe.angleX,Y,Z \\ - prevKeyframe.angleX,Y,Z)$$

And similarly for scaling. Each model would be translated to center, rotated to angleX,Y,Z for each respective axis and scaled to the new scale amount.

The camera transformations (saved as translation matrix T and rotation matrix R) are interpolated similarly.

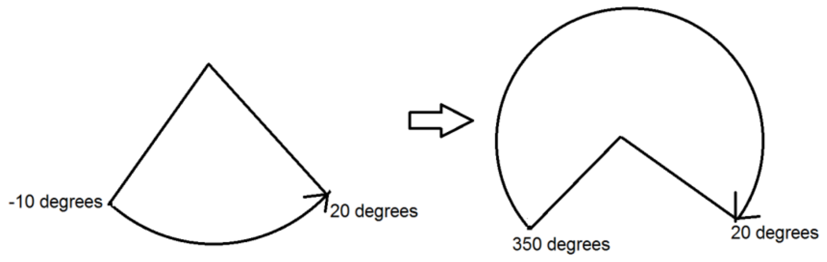
A problem arises when the angles of previous and next keyframes have different signs. Since all angles are saved as degrees between 0 and 360, a negative angle –*d* is saved as 360 – d. For example, instead of going from 20 to -10 degrees, the interpolation would make it go from 20 to 350 degrees!



This is problem is solved by instead interpolating the angle with:

$$angleX,Y,Z = prevKeyframe.angleX,Y,Z - t * (nextKeyframe.angleX,Y,Z - prevKeyframe.angleX,Y,Z - \mathbf{360}°)$$

Another challenge is determining whether the rotation is supposed to be clockwise or counter-clockwise. Each keyframe saves the field *Vec3 ccw* for each model, containing the total mouse displacement when rotating the model. If the *ccw* for the axis is negative, the model was rotated counter-clockwise, otherwise it's clockwise. To avoid a problem similar to previously described:



the angle would be interpolated instead as:

$$angleX,Y,Z = prevKeyframe.angleX,Y,Z - t * (nextKeyframe.angleX,Y,Z - prevKeyframe.angleX,Y,Z + \mathbf{360}°)$$

Keyframe Interpolation with Kinematic Equations

For models that move according to kinematic equations, rotations and scaling are the same as linear interpolation, but translations are interpolated according the initial velocity v_0, acceleration a to get from previous to next keyframe on time, and displacement d between the previous and next keyframes.

Each time a new keyframe is created, it updates its own v_0, and a and d of the previous keyframe for each model.

$$prevKeyframe.d = newKeyframe.center - prevKeyframe.center$$

$$t = newKeyframe.t - prevKeyframe.t$$

By rearranging kinematic equation $d=v_0*t + 0.5a*t^2$

$$prevKeyframe.a = \frac{2(prevKeyframe.d - t * prevKeyframe.v_0)}{t^2}$$

$$newKeyframe.v_0 = prevKeyframe.v_1 = prevKeyframe.v_0 + t * prevKeyframe.a$$

With v_0, a and d saved for each keyframe except for the final keyframe, we can interpolate based on kinematic equations to find the velocity and position of the model between two keyframes.

$$t = currentTime - prevKeyframe.time$$

$$center = prevKeyframe.center + t * prevKeyframe.v_0 + \frac{1}{2}t^2 * prevKeyframe.a$$

$$v = prevKeyframe.v_0 + t * prevKeyframe.a$$

This would allow models to change velocities smoothly, letting it make smooth turns and gradually speed up or slow down.

Motion after the Final Keyframe

After the final keyframe, the model can have 4 possible behaviors depending on its settings:

- **Linear interpolation without gravity** (Dynamics off, Gravity off): The model stops moving. Interpolation no longer happens for this model so it stays in the same place
- **Linear interpolation with gravity** (Dynamics off, Gravity on): The model falls. Interpolation happens to the model based on the acceleration of gravity G:
$$t = currentTime - finalKeyframe.time$$
$$center[downAxis] = center[downAxis] - \frac{1}{2}G * t^2$$
- **Dynamic interpolation without gravity** (Dynamics on, Gravity off): The model follows Newton's First Law and moves in constant motion based on v_0 of the final keyframe:
$$center = finalKeyframe.center + t * finalKeyframe.v_0$$
- **Dynamic interpolation with gravity** (Dynamics on, Gravity on): The model moves according to projectile motion, which is constant motion + gravity:
$$center = finalKeyframe.center + t * finalKeyframe.v_0$$
$$center[downAxis] = center[downAxis] - \frac{1}{2}G * t^2$$

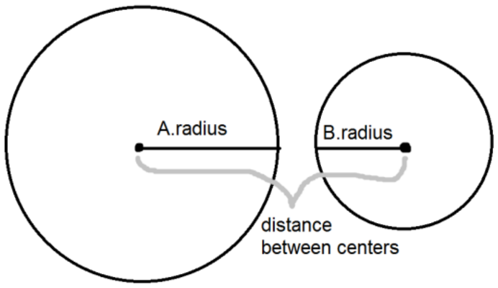
Collision Detection

Collision detection is optimized using bounding spheres [4]. Each model as a bounding sphere denoted by its center and radius. The center is the average of all vertices (sum of vertices / number of vertices) and the radius is the maximum distance from the center to any vertex.

For two models A and B, there is no collision if

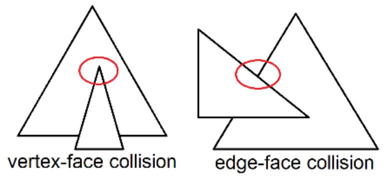
$$\sqrt{(A.center[x] - B.center[x])^2 + (A.center[y] - B.center[y])^2 + (A.center[z] - B.center[z])^2} > A.radius + B.radius$$

That is, the distance between the centers are greater than the sum of the radiuses means there's no collision.



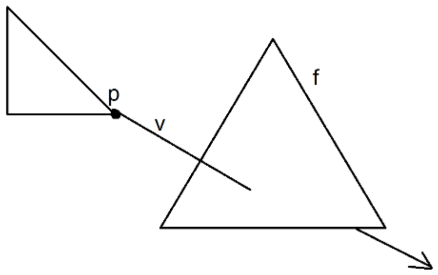
The broad phase runs efficiently because it doesn't need to check individual vertices of the mesh. If the broad phase does not detect a collision, the algorithm can return false. Otherwise, it has to run the narrow phase [2].

The narrow phase checks for vertex-face collision, face-vertex collision, edge-face collision and face-edge collision to determine if model A has collided with model B.



- vertex-face collision:

```
v = velocity of A
for each vertex p in A
     for each face f in B
          rayTriangleIntersection(ray(p, v), triangle(f))
          point of intersection = p + t*v
          if t<=0
               n = f.normal
               return collisionDetected
```



- face-vertex collision: Similar to vertex-face collision, but uses –v as velocity, checking all vertices in B with all faces in A, and uses the normal of B's vertex as the collision normal
- edge-face collision:
  ```
  for each edge e in A
       for each face f in B
            rayTriangleIntersection(ray(e), triangle(f))
            point of intersection = e.start + t*e
            if 0 <= t <= 1
                 n = f.normal
                 return collisionDetected
  ```
- face-edge collision: Similar to edge-face collision, but checking each edge in B with each face in A

Ray-Triangle Intersection [6] is done with:

```
rayTriangleIntersection(ray r, triangle f)
     if dot(f.normal, r) == 0 // r is parallel to f
          return false
     t = dot(f.point0 – r.start, f.normal) / dot(r, f.normal)
     if t<0 or t>1 // intersection outside of endpoints
          return false
     poi = r.start + t * r
     // check intersection point is in triangle
     if dot(n, cross((f.point1-f.point0), poi-f.point0)) >= 0
     and dot(n, cross((f.point2-f.point1), poi-f.point1)) >= 0
     and dot(n, cross((f.point0-f.point2), poi-f.point2)) >= 0
          return true
```

Since spheres are used very often and have a large number of vertices, I made the program more efficient by using a different algorithm [7] in the narrow phase if one of the models is a sphere to avoid checking all its vertices.

If A and B are both spheres, collision is detected if broad phase collision is detected. The normal would be normalize(A.center – B.center)

If only A is a sphere, I use a sphere-triangle intersection algorithm for the sphere A with every face in B

```
sphereTriangleIntersection(sphere A, triangle f)
     // check if A touches the plane of f
     if abs(dot(f.point0, f.normal)) > A.radius
          return false
     // check if A lies within the vertices of f
     for each vertex p in f
```

```
          // distance from p to A.center
          d = sqrt(dot(p – A.center, p – A.center))
          if d > A.radius
               and dot(f.point1 - p, p– A.center) > 0
               and dot(f.point2 - p, p– A.center) > 0
                    return false
     // check if A lies within the edges of f
     for each edge e in f
          t = dot(A.center – edge.start, edge) / dot(edge, edge)
          p = edge.start + t * edge
          if sqrt(dot(A.center – p, A.center – p)) > r
               and dot(A.center – p, f.oppositeVertex - p) < 0
                    return false
     return true
```

3D Elastic Collisions

When Dynamics is enabled for a model A, it will bounce off after colliding with model B instead of stopping. If B does not have Dynamics enabled, the new velocity v_1 of A will be the reflection of the velocity v_0 of A before the collision. It will be reflected about the normal n of the face in B that A collided with.

$$A.v_1 = A.v_0 - 2 * dot(A.v_0, n) * n$$

If B has Dynamics enabled, we do 3D elastic collisions.

In 1D elastic collisions, according to the conservation of momentum:

$$v_{a1} = \frac{m_a - m_b}{m_a + m_b} v_{a0} + \frac{2m_b}{m_a + m_b} v_{b0}$$

$$v_{b1} = \frac{m_b - m_a}{m_a + m_b} v_{b0} + \frac{2m_a}{m_a + m_b} v_{a0}$$

We can extend this to 3D by splitting the velocity into components

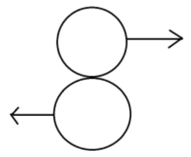
$$v = v_\perp + v_\parallel$$

where $v_\perp$ is the component of velocity of A that is perpendicular to the face of B and $v_\parallel$ is the component of velocity that is parallel to the face. This means:

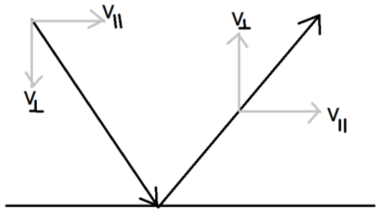
$$v_\perp = dot(v, n) * n$$

$$v_\parallel = v - v_\perp$$

The parallel component of velocity would be unaffected by the collision. Imagine two objects, one going left, the other going right, only scraping against each other. They will continue moving unaffected by that collision.



The perpendicular component of velocity will change according to 1D elastic collisions. We can treat it as 1D, because everything moves along the collision normal.



We get the equation for 3D elastic collision:

$$v_{a1} = v_{a0\parallel} + \frac{m_a - m_b}{m_a + m_b} v_{a0\perp} + \frac{2m_b}{m_a + m_b} v_{b0\perp}$$

$$v_{b1} = v_{b0\parallel} + \frac{m_b - m_a}{m_a + m_b} v_{b0\perp} + \frac{2m_a}{m_a + m_b} v_{a0\perp}$$

Sound

The collision and moving sounds are played with the SFML Audio library [8]. This library has APIs:

sound.play(): plays the sound

sound.pause()

sound.stop():stops at the beginning

sound.volume(float volume): changes the volume between 0 and 100

The sounds are loaded from WAV files into a SoundBuffer.

Every time there's a collision, each collided model plays its own collision sound. For the moving sound, this plays whenever the model has non-zero velocity.

The volume of the sound depends on distance to the camera and kinetic energy. The kinetic energy gets transferred into sound during the collision, or when it's moving and colliding with air particles to create sound.

The kinetic energy $E_k = \frac{1}{2}mv^2$

The volume (intensity) of the sound is affected by distance to camera according to the Inverse Square Law for Sound Intensity [5]: $I = \frac{P}{4\pi r^2}$, where I is the intensity of the sound, P is the source power, r is the distance to the sound's source

We get:

$$volume = c\frac{E_k}{4\pi r^2}$$

We multiply volume by a constant c to ensure that it's a reasonable value between 0 and 100, not too loud or too quiet, so that changes to E_k and r can be heard

Texture Mapping

Textures are loaded from BMP files. The program that loads the texture reads the bytes in the file. The header specifies information, such as the dimensions and size of the body. The body contains BGR values for every 3 bytes for every pixel, which are reversed to get RGB values.

We bind the texture to an index with `glBindTexture` and save that index in a list of textures. Then we initialize the texture to contain the data from the BMP file with `glTexImage2D`. Then, set min and mag filter to linear with `glTexParameteri`, which averages the pixel being displayed with the surrounding pixel colors when the display pixel maps to an area that's larger or smaller than a pixel in the texture.

When rendering a model, if it has a texture with id texid, we activate the texture with:

```
glActiveTexture(GL_TEXTURE0 + texid);
glBindTexture(GL_TEXTURE_2D, texid);
```

and pass the texid to the Fragment Shader.

In the Fragment Shader, the current pixel would have the color from

```
texture2D(texid, fs_in.uv).rgb
```

which obtains the color of the texture image at the pixel specified by the UV coordinates

The UV values of each vertex are read from the OBJ file of the model. They specify which part of the texture image each vertex maps to. For example

```
vt 0.0 0.0 #bottom left corner of the image
vt 1.0 1.0 #top right corner of the image
```

Each face in the OBJ file would then be denoted as

```
f a1/b1/c1 a2/b2/c2 a3/b3/c3
```

where `ai, bi, ci` are integers specifying which coordinate, uv value, normal are associated with vertex i in the triangular face.

The Vertex Shader interpolates the UV value at each pixel using the UV of the vertices, and passes this value to the Fragment Shader for texture mapping.

Skybox

The skybox is a cube around the camera with textures for each of the 6 sides. Only camera rotations are applied to the skybox when rendering, not camera translations. This gives the illusion that the sides of the box are very far away.

Also, before rendering, we set `glDepthMask(GL_FALSE)` to prevent the models from disappearing outside the skybox. This ensures that nothing ends up covered by the box. Everything is rendered in front of the box's rendering.

The 6 textures are mapped to their respective side of the cube, similar to texture mapping, but using `GL_TEXTURE_CUBE_MAP` instead of `GL_TEXTURE_2D`

The SkyboxFragmentShader gets the pixel's color with

```
texture(skyboxid, position)
```

where `skyboxid` is of type `samplerCube` passed from the program as a uniform, and `position` comes from the VertexShader describing the coordinates of the point being rendered

Reflection and Refraction Mapping

Reflection mapping lets the model reflect the skybox, while refraction mapping makes the model transparent.

In the Vertex Shader, the reflected and refracted rays are obtained through:

- Get the reflected ray using the position and normal in the view coordinate space, where the camera is the origin and the position_ES is the ray from the camera to the point

  ```
  vs_out.reflected = reflect(normalize(vs_out.position_ES),
  vs_out.normal_ES);
  ```

- undo the transformation that put the point into view coordinate space

  ```
  vs_out.reflected = vec3(inverse(ModelView) *
  vec4(vs_out.reflected, 0.0));
  ```

- get the refracted ray using the position and normal in the view coordinate space, where the camera is the origin and the position_ES is the ray from the camera to the point. refractionRatio = environmentRefractiveIndex / modelRefractiveIndex

  ```
  vs_out.refracted = refract(normalize(vs_out.position_ES),
  vs_out.normal_ES, refractionRatio);
  ```

- undo the transformation that put the point into view coordinate space

  ```
  vs_out.refracted = vec3(inverse(ModelView) *
  vec4(vs_out.refracted, 0.0));
  ```

The reflected and refracted rays get passed to the Fragment Shader. The Fragment Shader gets the color of the pixel based on the color of the skybox where the reflected/refracted ray hits.

```
reflectedCol = textureCube(skyboxMap, fs_in.reflected).rgb;
refractedCol = textureCube(skyboxMap, fs_in.refracted).rgb;
```

It takes a percentage of the reflected or refracted color based on the reflectivity or transparency settings of the model. The rest of the color is the diffuse color or texture. For example:

```
totCol = col * (1 - reflectivity) + reflectedCol * reflectivity;
```

Cel Shading

Cel shading is done in the Fragment Shader by taking the color before it gets outputted and snapping it to an interval through a step function. For example, after getting fragColor through the phongModel (the color comes from kd, ks, texture or reflection/refraction mapping), we run:

```
for (int i=0; i<3; i++) {
    fragColour[i] = step10(fragColour[i]);
}
```

That is, for every each value in RGB for the fragment color, snap it to an interval with

```
float step10(float n) {
    for (int i=1; i<=10; i++) {
        if (n < i/10.0) return i/10.0;
    }
}
```

This sets values less than 0.1 to 0.1, values between 0.1 and 0.2 to 0.2, values between 0.2 and 0.3 to 0.3, and so on

Importing and Exporting Animations

The keyframes and settings of an animation can be saved in .ani files, which are basically text files that the program can parse. This would allow users to replay existing animations and edit animations over a long period of time.

Exporting an animation writes to a file named Animations/new.ani. It outputs keyframe and model information into the file, such as the NodeState for every model in every keyframe, the texture that each model uses, which skybox being used, etc.

When importing an animation, the program reads the file into variables such as the list of keyframes, or the fields saving the model settings. It expects all the data to be in the same order and locations that it was exported as.

Future Possibilities

Reflecting Other Models

Right now, a model with reflectivity only reflects the skybox, not other models around it. The reflected ray from the Vertex Shader is used by the Fragment Shader to determine the color of the skybox where the reflected ray hits. Instead of using the skybox's texture, the program could use other models' textures, determine which model the reflected ray will hit and use the color from the other model's texture instead of the skybox. The Fragment Shader would need to be passed the textures of every model. There would need to be an algorithm in the Fragment Shader for determining which model the reflected ray hits first

Animating models with joints

Right now, models with joints will not move the joints when the animation plays, because it thinks it's colliding with the node above it in the node tree. This should be easy to fix by only doing collision detection for nodes on different subtrees of the root node, not for nodes on the same subtree.

Inelastic Collisions with Deformation [3]

In an inelastic collision, some of the kinetic energy gets transformed into energy to make the model deform. To do deformation, each vertex in the mesh has to be transformed, rather than transforming the whole model as a rigid body. When a vertex collides with a surface, it would stop while the other vertices keep moving until they also collide. This would cause the model to squish when it hits the surface. When it reaches a certain point based on the viscosity of the model, it might bounce back. The squished vertices would return to its original place in the model when it no longer collides with the surface.

Torque

When two models collide it should affect their rotations along with their translations. When a rotating model hits another model, it would cause itself the other to rotate in the opposite direction. When a model scraps against another model, they should start rotating. For the physics engine to simulate torque, the models would need to have friction. It would get the component of angular velocity from the velocity before collision to determine how fast the models will rotate.

Bibliography

[1] Kort, Alexander. “Computer Aided Inbetweening.” Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering - NPAR '02, 2002, doi:10.1145/508530.508552.

- This paper provides algorithms for interpolating between keyframes. Although it focuses on 2D animation, the concepts can still be transfered to 3D

[2] Moore, Matthew, and Wilhelms, Jane. “Collision Detection and Response for Computer Animation.” Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '88, 1988, doi:10.1145/54852.378528.

- This paper provides equations for detecting if two models have collided and how they should behave afterwards.

[3] Barbic, Jernej, et al. “Deformable Object Animation Using Reduced Optimal Control.” ACM SIGGRAPH 2009 Papers on - SIGGRAPH '09, 2009, doi:10.1145/1576246.1531359.

- This paper describes how to do physically based animations, focusing on deforming the model rather than moving the model through the scene

[4] Sulaiman, Hamzah Asyrani, and Abdullah Bade. Bounding Volume Hierarchies for Collision Detection. INTECH Open Access Publisher, 2012.

- This textbook describes how to use bounding volumes to make collision detection more efficient for complex models

[5] “Inverse Square Law, Sound.” Inverse Square Law for Sound, http://hyperphysics.phy-astr.gsu.edu/hbase/Acoustic/invsqs.html.

- Describes how sound intensity varies with distance

[6] Scratchapixel. “Ray-Triangle Intersection: Geometric Solution.” Ray Tracing: Rendering a Triangle, 15 Aug. 2014, www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.

- Algorithm for detecting intersection between line and triangle, used in collision detection

[7] Ericson, Christer. “Optimizing a Sphere-Triangle Intersection Test.” Realtimecollisiondetection.net – the Blog, 30 Dec. 2010, http://realtimecollisiondetection.net/blog/?p=103.

- Algorithm for detecting intersection between sphere and triangle, used for optimizing collision detection for spheres

[8] Gomila, Laurent. “Playing Sounds and Music.” SFML Tutorials, www.sfml-dev.org/tutorials/2.4/audio-sounds.php.

- Tutorial on how SFML Audio works, the library used to play sound