CS 4621 PPA3: Animation

out: Saturday 19 November 2011
due: Friday 2 December 2011

1 Introduction

There are two parts to this assignment. In the first part, you will complete the implementation of key frame animation, by implementing interpolation for rotations, translations and scaling. The second part of the assignment is more open-ended; you will create time-varying ("animated") shaders for fire, clouds, and a third shader of your choice.

2 Key frame animation

In this first part of the assignment we have implemented a key frame animation framework (as described below). You will need to build on that framework and implement a few main features that enable correct interpolation of animations between keyframes.

2.1 The interface

We have added a new panel "Animation" panel, which contains different controls for animation, briefly described here:

- Jump to frame: this lets you drag the slider to the frame you want, or enter the number of the frame directly in the textbox.
- Add a keyframe: jump to the frame where you want to make a keyframe, then click the button "Keyframe it".
- Remove a keyframe: select a keyframe and right click on it, choose "Remove Keyframe" from the popup menu. Frame 0 cannot be removed.
- Play the animation: click the button "Play!" to start animating from the current frame. Click it again to stop the animation.
- Interpolate with cubic splines: select the checkbox "Cubic spline interpolation".
- Capture frames to file: check "Capture to file" and click "Play!", all the subsequent frames being animated will be exported to file.

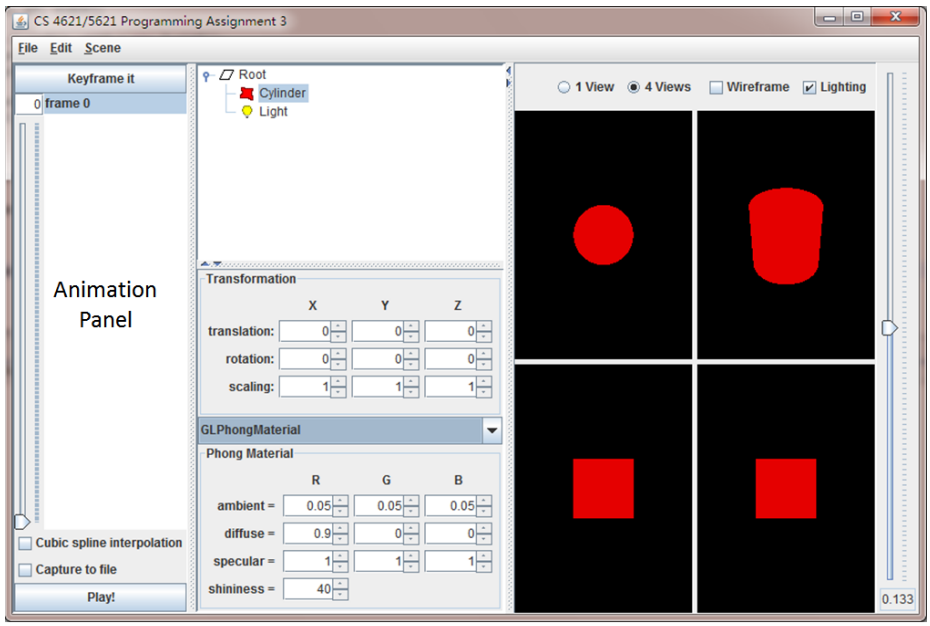

Figure 1: Interface

- Change shader: click some mesh from the tree hierarchy, then select one of the shaders below.
- Open/Save all keyframes as a file: click File->Open/Save As, then select the file to Open/Save. Warning: all keyframes should have the same scene hierarchy.

2.2 Functionality to implement

You will fully implement all the functionality needed to make the animation work in the framework described above. There are two possible interpolation options: linear and cubic. Both of these interpolations are possible for translation and scaling. For rotations you will only implement spherical linear interpolation (not cubic).

2.2.1 Implementing interpolation

Interpolation is done in `AnimationController`. Keyframes are stored in `TreeMap<Integer, Scene> keyframes`. You will need to implement:

- `Scene getLinearlyInterpolatedScene(int frame)` and
- `Scene getCatmullRomInterpolatedScene(int frame).`

to return the interpolated scene for a given frame number.

`getLinearlyInterpolatedScene` should return a linearly interpolated `Scene`. Similarly, `getCatmullRomInterpolatedScene` should return a `Scene` interpolated using the Catmull-Rom cubic spline for translation and scaling. You will always interpolate rotation using spherical linear interpolation, **slerp**. While vecmath does implement the **slerp** functionality, you will write your own code to implement it.

For `getLinearlyInterpolatedScene`, you need to:

1. If the current frame is at some keyframe, we can return that keyframe and don't need to interpolate.
2. Otherwise, find the two `Scenes` which have the nearest bigger and smaller frame numbers than the current frame.
3. Create a new `Scene` with the same structure as them. To duplicate the `Scene` structure, one needs to use `DuplicateVisitor`.
4. Flatten the three `Scenes` to an array. Use `FlattenVisitor` to achieve this.
5. Traverse the arrays and interpolate the translation, rotation and scaling components of each `TransformationNode` from the two keyframed `Scenes`.

For `getCatmullRomInterpolatedScene` the technique is the same, except you will use cubic, instead of linear interpolation.

2.2.2 Catmull-Rom spline interpolation

The Catmull-Rom spline is a cubic interpolating spline that goes through all its control points.

$$P(t) = \frac{1}{2}\begin{bmatrix}1 & t & t^2 & t^3\end{bmatrix}\begin{bmatrix}0 & 2 & 0 & 0\\ -1 & 0 & 1 & 0\\ 2 & -5 & 4 & -1\\ -1 & 3 & -3 & 1\end{bmatrix}\begin{bmatrix}P_{i-2}\\ P_{i-1}\\ P_i\\ P_{i+1}\end{bmatrix}$$

where P_i's are control points. In addition, it has the property that:

1. $P(t=0) = P_{i-1}, P(t=1) = P_i$
2. Further the tangents at each point are defined by the next and previous point as follows:

$$v_{i-1} = 0.5(P_i - P_{i-2})$$
$$v_i = 0.5(P_{i+1} - P_{i-1})$$

2.2.3 Finding control points

To interpolate frame x using the Catmull-Rom spline, we need to find four frames to define four control points. We call them x_0, x_1, x_2 and x_3, so that $x_0 < x_1 < x < x_2 < x_3$. These frames are:

- x_1 is the nearest keyframe smaller than x
- x_0 is the nearest keyframe smaller than x_1
- x_2 is the nearest keyframe bigger than x
- x_3 is the nearest keyframe bigger than x_2

There will also be some corner cases. When x is larger than the last keyframe, we can duplicate the state of the last keyframe. When less than four control points can be found, we will use some of them twice (e.g., assuming we have three keyframes P_0, P_1 and P_2, and we want to interpolate frame x, $P_0 < x < P_1$, then the four control points will be P_0, P_0, P_1, P_2). In this case the spline will still go through the keyframes. The convention described here to process corner cases for cubic interpolation also applies to linear interpolation.

2.2.4 Conversion between Euler angles and Quaternions

Because we are going to implement **slerp**, which uses quaternions, while the rotation manipulator uses Euler angles, we need to convert between these two representations.

From Euler angles to Quaternions

The euler angle (a, b, c) is the product of three axis-angles given us:

$$< (1, 0, 0), a >$$
$$< (0, 1, 0), b >$$
$$< (0, 0, 1), c >$$

So they represent the product of three quaternions:

$$q_x =< s, v >=< \cos(a/2), \sin(a/2)(1, 0, 0) >$$
$$q_y =< s, v >=< \cos(b/2), \sin(b/2)(0, 1, 0) >$$
$$q_z =< s, v >=< \cos(c/2), \sin(c/2)(0, 0, 1) >$$
$$q = q_z q_y q_x$$

From Quaternions to Euler angles

To convert from quaternion $< w, x, y, z >$ to Euler angle (a, b, c) we need the following equation

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \text{atan2}(2(wx + yz), 1 - 2(x^2 + y^2)) \\ \arcsin(2(wy - zx)) \\ \text{atan2}(2(wz + xy), 1 - 2(y^2 + z^2)) \end{bmatrix}$$

(Reference: Wiki page "Conversion between quaternions and Euler angles")

2.3 End product

Once you have implemented these components you will be able to set up key frame animation, interpolate to create the inbetween frames, and save animations away. We would like you to use your engine to create a couple of interesting examples (e.g., link and joint structures) to show off your engine. Turn the animations in as well. Also include a README describing what you created.

3 Shaders

You will implement 3 time varying shaders in the assignment: a fire shader, a cloudy sky shader, and one other animated shader of your choice. This portion of the assignment has a lot of freedom; we broadly describe the first two shaders here, and encourage you to play around and make something fun for the third.

3.1 Procedural Noise

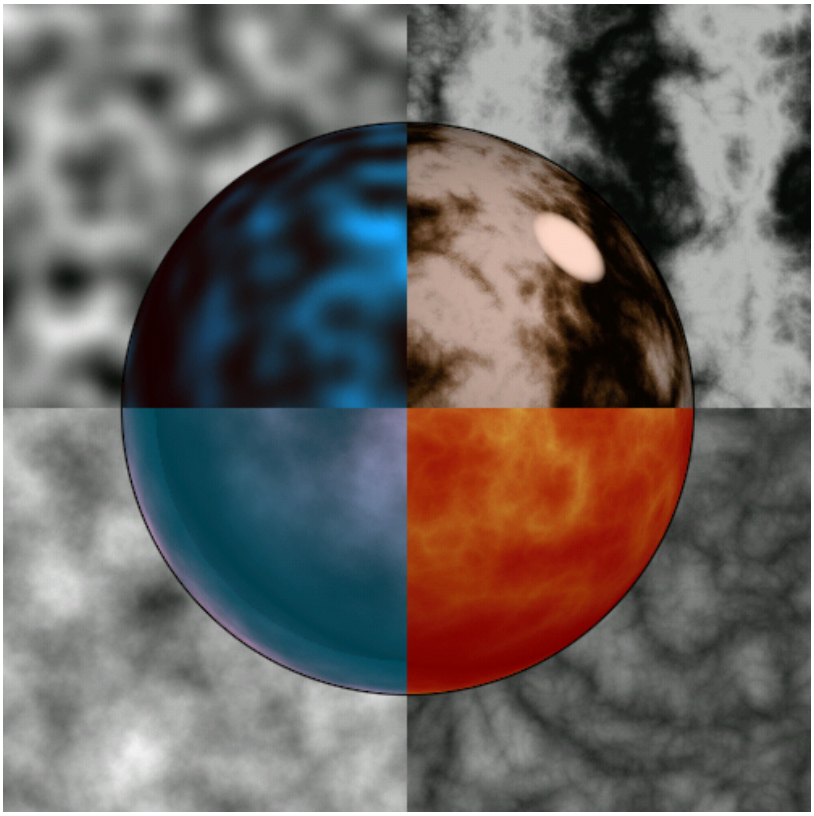

Figure 2: Some examples of procedural textures based on Perlin noise. You will use this type of noise in your animated shaders to provide unpredictable movement (flickering of a fire, etc.). Image from a 1999 talk by Ken Perlin, available online at `http://www.noisemachine.com/talk1/`.

To simulate the unpredictable flickering of a fire or the evolution of clouds, we need a source of randomness. Truly random numbers (from `rand()`) aren't appropriate because fire does not move in a completely random way; it moves unpredictably, but the motion is smooth and continuous. Of course, people have thought about this problem before. Perlin noise, invented in 1984 by Ken Perlin for exactly this task, is a robust and commonly used method for generating smooth randomness. Fig. 2 shows some examples of the kinds of textures which can be created using Perlin noise. Generating Perlin noise is not difficult, but it's not the point of this assignment, so we have provided a texture of pre-generated noise for you to sample from in your shaders. Each component of the texture (RGBA) contains an independent noise distribution. Each successive channel has twice the frequency and

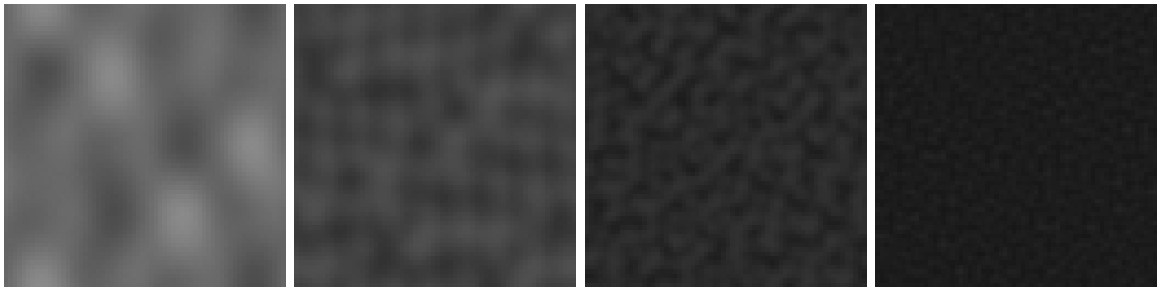

Figure 3: Red, green, blue, and alpha channels (left to right) of a single slice of the provided noise texture. You can see that each channel varies more quickly than the previous channel, but is half as bright.

half the amplitude of the previous channel, i.e. `noise.g` varies twice as fast as `noise.r` but is half as bright, and so forth. Fig. 3 shows each channel from a single slice of the 3D noise texture to illustrate this.

3.2 Fire Shader

The fire shader takes as input the noise texture we created for you, and animates a "fire-like" effect. Since we need a different noise value at every point in space, the Perlin noise texture we provide is a 3D texture. 3D textures are directly analogous to regular 2D textures: they are 3-dimensional blocks of pixels (sometimes called "voxels" for 3D textures), and you need 3 texture coordinates to sample from them. For this shader, you'll use the fragment's position in object space to sample from the noise volume. This gives a random `vec4` associated with that fragment, but it will be the same random value every frame; so far, this does not animate. The simplest way to animate is to add a time-dependent offset to one of the sample coordinates, i.e. `sample_noise(pos + vec3(0.0, time, 0.0))`. However, this produces a boring "scrolling" effect which doesn't look much like flame. You can create a more turbulent effect by recursively sampling from the noise volume, something like `sample_noise(pos + sample_noise(pos))`.

Once you have a sample of noise with suitable motion over time, you need to turn it into the color and texture of fire. This involves a lot of playing around to see what looks good, but here are some tips to get started:

- The provided shader has two constant colors, FireColor1 and FireColor2. You should use the noise texture to come up with some intensity value, and then interpolate between these two colors (GLSL has a `mix(value0, value1, t)` function) to get the final output color.
- You can sum different frequencies of noise with different amplitudes to create more visually complex images. For example, the lower left square of Fig. 2 has at least two contributions: the low-frequency high-amplitude noise which creates the overall light and dark areas, and the high-frequency but low-amplitude speckles. If you think about how fire looks, it has large-scale, smooth changes between "lots of fire" and "no fire", but intricate detail in the areas of "lots of fire". You can sum different channels from the noise texture (containing different frequencies, see Fig. 3) to simulate this.
- Since the noise values are sampled from a color texture, they are all in the range [0, 1].

Something like `2.0 * abs(noise - 0.5)` causes the derivative of the noise to change abrupty whenever the original noise crossed 0.5 , creating creases at such points (think about the graph of `abs(x)` — it has a tip at 0). You can play with the crease location (0.5 in this example) and amplitude (1.0 in this example) to create different effects.

The fire effect should fade out in the -y direction, so that the object is black at y = -1 and the effect is fully visible at y = 1. Since all the meshes generated in this assignment fit in a $2 \times 2 \times 2$ bounding box, you can just use the limits [-1, 1] in model space without having to tell the shader which object is being shaded.

3.3 Cloud Shader

The cloud shader also uses the noise texture, but creates the effect of clouds moving or fading in and out. This shader will be quite similar to the fire shader. Clouds are less turbulent than fire, so you probably won't use recursive sampling of the noise texture in this shader, but the other tips from the fire shader still apply. The cloud shader should implement basic diffuse lighting as well, using the light position queried from OpenGL and the generated cloud pattern as the object's diffuse color.

3.4 Open-Ended/Fun Shader

Anything you want! You will create a third shader of your choice. The only requirement is that it use Perlin noise to create an interesting animated effect. Perlin noise is used for everything and has been around for a long time, so there are tons of resources and ideas available online. You should also look up the list of GLSL built-in functions and just play around.

In the README file give us detailed descriptions of what your shader does/tries to do so that we can understand it. If you used online resources, or talked to your friends, make sure you tell us about your source.

4 Extra credit

For extra credit, go crazy creating interesting animated shaders!