eecs 487

INTERACTIVE COMPUTER GRAPHICS

EECS487 PA4: Textures, Splines, and Animation

The assignment is due on Friday, Dec. 5th, 2014 at 11:59PM.

Plain Page for Printing

Overview

In this assignment you will extend the `view3D` from PA2 to render animated textured scenes.

The given support code parses an input X3D file, partially constructs a scene graph, inserting various X3D nodes into it, and then passes the scene to OpenGL for rendering. The X3D file format is described in the X3D Specification. Most tasks require modifying the `scene.cpp` file (as usual, look for ``YOUR CODE HERE''). As part of your assignment you will need to create an example X3D scene file that will show an animated humanoid character moving through a scene.

Tasks

The first three tasks of this assignment familiarize you with how texturing fits into a scene graph. The first two tasks should be easy peasy for you by now. The third one introduces you to texture transform. When you build the provided code as is, you will likely get warnings about the `index` variable being not used in various functions in `scene.cpp`. You can ignore these warnings for now. The `index` variable will be used by your code. As is, `view3D` will display `scenes/aclock.x3d` as shown on the right.

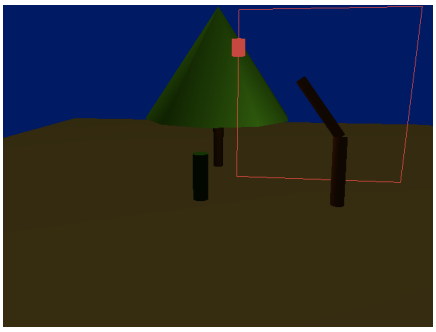
1. Texture Setup. The provided `view3D` application can load texture image from a JPEG or PNG file and store it as an object of `Image` class (`image_` points to the image). To enable texturing, you need to generate a new OpenGL texture handle (ID) and set up OpenGL for texturing. Modify the `SetupTexture` method of the `X3ImageTexture` class. You MUST enable mipmapping to avoid aliasing effects. Now `scenes/aclock.x3d` should look like on the right. Also check that `fish-textured.x3d` shows the fins striped. Image files used for texturing MUST be in the same folder as your scene file. **5 points**

2. Textured Cylinder. When rendering geometry, you need to specify texture coordinates for each vertex. Modify the `X3Cylinder::Render()` method to render a textured cylinder. Texture coordinates MUST be assigned as specified in the `X3Cylinder` spec. You may want to consult the `XBox::Render()` and `XCone::Render()` methods for examples. Now `scenes/aclock.x3d` should look like on the right (notice the upside down clock face). You should also be able to view `scenes/cylhead.x3d` now. **10 points**

3. Texture Transforms. OpenGL can apply tranformations to specified texture coordinates before using them. Modify the `X3TextureTransform::Render()` method to enable texture coordinates transformations. You MUST implement it as described in the `X3TextureTransform` spec. This is similar to what happened in the `X3Transform` node; however, there is no hierarchy and no need to use the matrix stack. Just override the transforms every time `X3TextureTransform` is needed. Now `scenes/aclock.x3d` should look like on the right (notice the corrected clock face). Also check that `wall.x3d` shows the correct mountain panorama. **10 points**

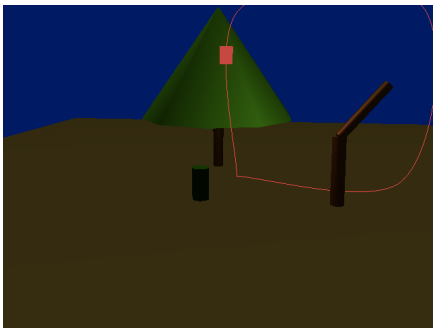
4. Linear interpolation. The provided support code has some basic animation capability in the `Timer`, `Interpolator`, and `Link` nodes. Each `Link` specifies which `Timer`'s time to feed into which `Interpolator`, to update which field of which node (see the section on *Field Update Mechanism* below).

Study the provided code that implements piecewise linear interpolation for the `X3ScalarInterpolator` and the `X3PositionInterpolator` classes. Given an array of keys, a corresponding array of values, and a time value, the code calls the `X3InterpolatorNode::FindKeyInterval()` method to determine the index between which two keys the time falls. The code computes an in-between time distance between the two keys and returns a linear interpolation of the values associated with the two keys. Due to polymorphism, the code for `X3ScalarInterpolator::LinearInerpolation()` and `X3PositionInterpolator::LinearInerpolation()` are identical. For rotations, angles and axes must be interpolated separately. Implement linear interpolation for `X3OrientationInterpolator` that linearly interpolates the axes and angles separately and returns the two together as a `rotation_t`. View `scenes/spline-linear.x3d`. Hit SPACEBAR to start the animation. All four objects in the scene should animate, each using a separate `Interpolator`. Study the file `scenes/spline-linear.x3d` to see which object uses which `Interpolator`. The scene `scenes/aclock.x3d` also animates. This task is mainly about getting you to spend the time to read the spec on *Field Update Mechanism* and to study the codes that implement linear interpolation and to study the scene file so that you're familiar with the support code structure for animation. Make sure you're thoroughly familiar with the spec and code and scene file before you move on to the next two tasks. **5 points**

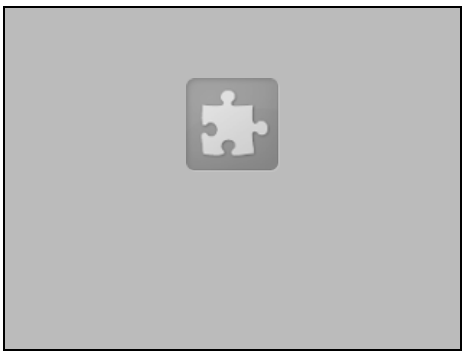


5. Spline Interpolation. Implement spline interpolation for the `X3ScalarInterpolator` and `X3PositionInterpolator` classes to enable animated content rendering--with smoothly varying parameters, as well as to enable rendering of smooth `X3Curve`. Use the Catmull-Rom splines (with tension zero) to produce smooth interpolation. You only need to implement smooth interpolation for the scalar and position values, not rotations (which would require use

of quaternions). You may adapt your code from Lab8 to this task. For easier debugging, use the `X3Curve` node to draw a curve in space. (`X3Curve` is not part of standard X3D.) The `X3Curve` has an `X3PositionInterpolator` as a child. It draws a curve in space that samples the `X3PositionInterpolator`'s trajectory. Once you have these implemented, you should be able to view all objects in the scene `scenes/spline-smooth.x3d` animated as in the previous task, but smoother. **15 points**



6. Humanoid Walking. Create an X3D scene file that renders an animated humanoid walking through some scene. For full credit, the character should be textured, have two legs (comprised of upper legs, lower legs, and feet), two arms, a body, a head, and exhibit a proper walk cycle. The walking should propagate the character through the scene at the appropriate rate without sliding. It is acceptable to use cylinders and boxes for rendering the character. This is 55% of the assignment; be sure to treat it as such!



The animated humanoid must be modeled as an articulated hierarchical character model. You must tie together several `Interpolation` nodes with proper rotation transformations of the skeletal links to produce a walking figure. Once a basic periodic walk cycle is created, you can propagate it through a scene with a translation transform. (As an optional extension, i.e., not required, walk your character on a curved path, which would mean changing its orientation also). Texture your character with some images to help indicate that you have control over your scene viewer (i.e., the head should point forward and knees bend backward). Do not worry about being overly realistic; think of your character as a puppet/robot (though the character parts should not fly off in different directions during animation). You MUST name your scene file `walk.x3d`. On the right is an example. Be creative and don't create a scene similar to this one! The best way to figure out enough X3D syntax to complete this task is to study `spline-smooth.x3d`. Some students have also found this X3D tutorial useful. **55 points**

Field Update Mechanism

Overview

We use a simplified custom mechanism for animating node properties. A special scenegraph node, `Link`, is introduced to bind together three components required for animation, namely, *timers*, *interpolators*, and *fields to be updated*. Specifically, each `Link` stores the pointer to the field to be updated, and it references the `Timer` and `Interpolator` nodes. In each frame, before the rendering pass, each `Link` updates the value of the field by the value produced by the `Interpolator`. Each `Interpolator` node has its `<key, value>` sequence as attributes read from the X3D file.

Each `Interpolator` may be used in multiple instances: for example, a pendulum orientation interpolator may be reused in several different clock gadgets present in the scene. All of these gadgets would go through the same sequence of orientations, even if the gadgets have different speeds and

are not synchronized with each other. The gadgets use the same `Interpolator` but different `Timer`s to drive the `Interpolator`. Thus, for each gadget a separate `Link` node will be used to refer to the same `Interpolator` but different `Timer`s. The role of the `Timer` is to take a current global time and perform a timing transformation on it. Schematically, a single update operation for a `Link`, `L`, can be represented as: `L.field_pointer = L.interpolator.Evaluate(L.timer.ConvertTime(t))`, where `t` is the current global time. Global time runs in sync with the system clock, however it may be paused and restarted by the application.

Timers

`Timer` nodes define time conversion from global time to a warped time used as a parameter to the interpolator. A timer has two attributes: period and shift.

`Timer`s with positive periods are periodic, they produce values between zero and one. Therefore, in order to produce periodic motion, one should define `Interpolator`s whose key sequence starts at zero and ends at one. To avoid abrupt transitions at the end of each cycle, the periodic `Interpolator` should have the same starting and ending values in its value sequence.

A `Timer` with a negative period attribute is non-periodic, it simply scales the input time by 1/(-period). This can be used with `Interpolator`s that have no restrictions on their key sequence (as long as it is increasing).

Please refer to `X3Timer::ConvertTime()` method for more information.

Interpolators

An `Interpolator` maps its input time into a value. The parameters of the mapping are the key and value sequences specified in the corresponding attributes in the X3D file. Depending on the output value, we define three interpolator classes:

- Scalar interpolators (return a single float value)
- Position interpolators (return a three-dimensional vector of floats, i.e., `XVec3f`)
- Orientation interpolators (return a rotation value, defined in `rotation_t` struct)

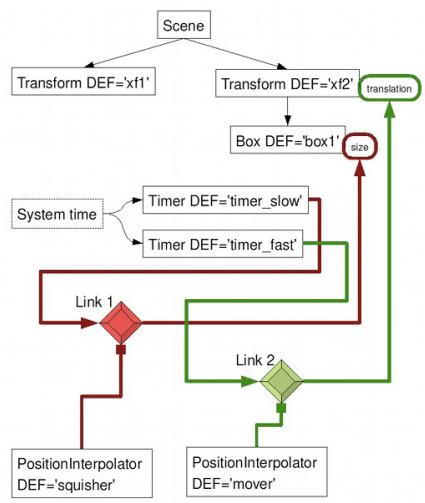
Depending of the value of the *smooth* flag, the `Interpolator`s implement piecewise linear or cubic Catmull-Rom spline interpolation. Orientation `Interpolator`s will not use smooth interpolation (which requires use of quaternions).

Link nodes

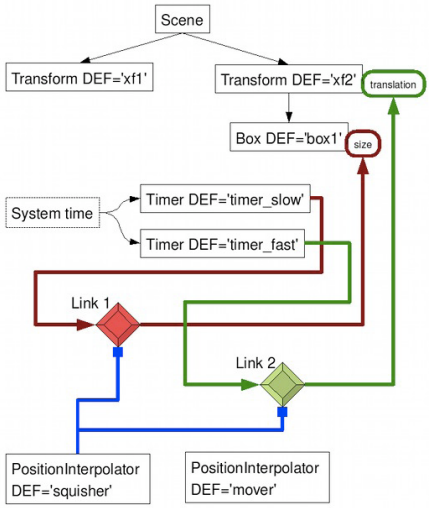
A `Link` node can appear anywhere within the scene, and has four required attributes:

- `TIMER` refers to a timer's name defined earlier in the scene file
- `INTERPOLATOR` refers to an interpolator's name defined earlier in the scene file
- `TO_NODE` refers to a node whose field needs to be updated in every frame by the output of the `Interpolator`. Again the attribute should contain a name of such node defined earlier in the scene file. In order for a node to accept values into one of its fields, it needs to process the field's name using its `GetFieldPointer` method (see the `X3Transform::GetFieldPointer()` for an example).
- `TO_FIELD` refers to a field's name within the updatable node declared with a `TO_NODE` attribute.

(The `Link` node is not part of the X3D standard. It is introduced in this assignment to define update relations.) The diagram below shows an example arrangement with two links using two different `Timer`s, two `Interpolator`s, and two updated fields.



In some situations, we can reuse `Interpolators` or `Timers`. For example, in the following diagram, the two `Links` use the same `Interpolator` but different `Timers`. Therefore, the same sequence of 3D vector values will be placed into the two updated fields, possibly with some delays. Or perhaps the values in the translation field of the transformation node will change more quickly (since it is using the "`timer_fast`" `Timer` which may indicate a shorter period value). In a similar fashion, a single `Timer` can be used to drive two different `Interpolators`.



Support Code and Required Libraries

Download the assignment archive from `/afs/umich.edu/class/eecs487/f14/FILES/pa4.tgz`.

The program `view3D` should be called with one command-line argument--the name of an X3D scene file. We have included a few in the `scenes` sub folder. The code needs to be linked against the `png`, `jpeg`, `expat` and the usual OpenGL, GLU, and GLUT libraries; and its compilation requires the corresponding header files. For instructions on how to install these libraries, please see the course note. Your code must not require other external libraries or include files other than the ones included in the support code or listed in this spec (e.g., do not include `xmat.h`).

The following keyboard shortcuts are defined for `view3D` (you may bind other operations to keys in `view3D.cpp:kbd()`, but do not change these predefined keys):

- 'q' or ESC: quits the program
- SPACE: pauses/resumes animation
- RIGHT_ARROW: moves one frame forward (when the animation is paused)
- LEFT_ARROW: moves one frame back (when the animation is paused)
- HOME: moves to time zero (when the animation is paused)

You can rotate the object around by left-click-and-hold while dragging the mouse about. Dollying in and out can be done by right-click-and-hold while dragging the mouse about.

Submission Guidelines

As with PA1, to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. For example, if the assignment asks you to implement sort using heap sort and you turn in a working program that uses insertion sort in place of the heap sort, it will be considered cheating. If you can not implement a required algorithm, you *must* inform the teaching staff when turning in your assignment, e.g., by documenting it in your writeup.

The creative portion of this project, Task 6, will take some time to do right if you are interested in getting a good score. It involves a fair amount of trial and error.

Create a writeup in **text format** that discusses:

1. Your platform: Mac OS X, Linux, Windows, or any other.
2. Anything noteworty about your implementation.
3. Feedback on the assignment.
4. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Your "*PA4 files*" then consists *only* of your `writeup-uniqname.txt`, `scene.cpp`, your humanoid walking scene file from Task 6 (which MUST be named `walk.x3d`), and new texture file(s) you use as part of Task 6.

To turn in your PA4:

1. Email the GSI the SHA1's of your *PA4 files*. Use "EECS487: PA4 Submission" as your email's "Subject:" line. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. `scp` your *PA4 files* to your `pa4` folder on IFS:
   `/afs/umich.edu/class/eecs487/f14/FOLDERS/<uniqname>/pa4/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Or you can upload it by pointing your web browser to mfile. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 email will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. Try not to email your SHA1 to the GSI until you've finalized your code. You don't want to annoy him.

Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object, executable, dll, library, or image files) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s you've added for debugging purposes.

General Information

The General Information section from PA1 applies. Please review it if you haven't read it or would like to refresh your memory.