CG Programming I – Assignment #2 (Lit cube scene)

19-October-2011

In this assignment, you will implement a simple scene containing several lit, animated cubes. This assignment is divided into several parts. Each part is due in successive weeks.

1 Support Routines - due 26-October-2011

In the first part, you will implement a series of routines that will form the basis of the remaining parts. All of this will be implemented in C / C++ code.

- Using the provided `GLUvec4` and `GLUmat4` classes, implement the following functions:
  - `rotate_x_axis` - Calculate a matrix that rotates around the X axis by some specified angle.
  - `rotate_y_axis` - Calculate a matrix that rotates around the Y axis by some specified angle.
  - `look_at` - Calculate a basis matrix from an eye position, a "look at" position, and an up direction.
  - `perspective` - Calculate a perspective projection matrix given a field-of-view angle (for the Y dimension), an aspect ratio, and near and far plane distances.

  You may use the multiplication, addition, dot-product, and cross-product functions provided by the GLU3 library. You may also use the translation matrix (`gluTranslate`, etc.) functions. The code for these functions is available in `glu3_scalar.h`. You may look at this code if you wish. You *may not* use the rotation functions (`gluRotate4v`, etc.), look-at functions (`gluLookAt4v`, etc.), or perspective matrix functions (`gluPerspective4f`, `gluFrustum6f`, etc.).

  As you implement the matrix operations, implement unit test to verify the results. For example, the rotation routines should produce predictable results at $0°$, $90°$, $180°$, $270°$, and $360°$. The `look_at` function can be verified by comparing its result with the result of several simpler transformations (e.g., a series of rotations and translations) that are composed together. The test functions should live in separate files and should have names like `check_rotation`, `check_cube`, etc. These functions should *always* be called from `main` as early as possible. This will help identify any bugs that you may introduce later.

  It is strongly advisable, though not required, to implement the unit test *before* implementing the functions that they test. Without an implementation, the unit tests should all fail. This technique is called *test-driven development*[1].

2 Simple Scene - due 2-November-2011

The first part requires only a single cube rotating in the scene.

- Implement a routine that creates a buffer object and fills it with the vertexes of a cube. This code should use the `GLUcube` class provided in the GLU3 library. You will need to implement a subclass of `GLUshapeConsumer` to receive data from the `GLUcube`.
  - Decide how to store vertex data for the cube. What data will be sotred for each vertex? What format will be used?
  - Decide how to store the element index data for the cube. What format will be used?
  - Use `GLUshape::vertex_count` to determine how much space is needed for the vertex data.
  - Use `GLUshape::element_count` to determine how much space is needed for the element index data.
  - Create a buffer object with sufficient storage to hold *all* of the data.
  - Use `GLUshape::generate` with your class derived from `GLUshapeConsumer` to store the data in the buffer object.

[1] `http://en.wikipedia.org/wiki/Test-driven_development`

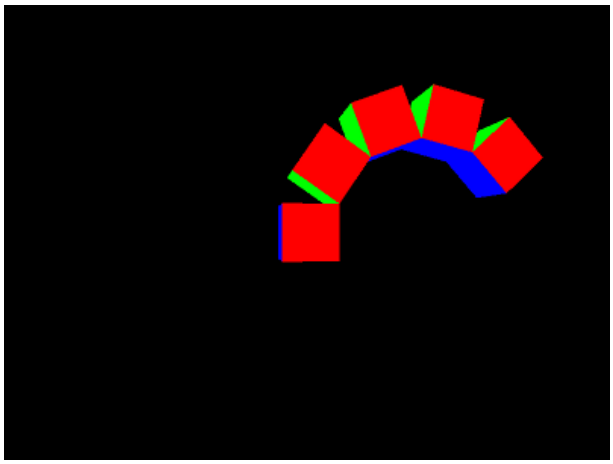

Figure 1: Arch of rotating cubes

- Implement a vertex shader that will transform incoming vertices by a model-view-projection matrix. The vertex shader should also pass a per-vertex color through to the fragment shader.
- Implement a fragment shader that takes a color from the vertex shader and emits it as the fragment color.
- Implement a display routine that will render the cube rotated by some angle. `glDrawElements` will be used to draw the data generated in the previous step. *Do not use* any other drawing function. The angle of rotation varies by time. Pick some rotation speed, say 30° per second, and use the elapsed program time to deterine the rotation angle each frame.

3 Complex Scene - due 9-November-2011

The second part requires several additions. Not only is the scene more complex, but a simple culling algorithm must be implemented.

- Instead of a single cube, five cubes must be rendered. The cubes will start stacked in a column. Each cube will rotate around the edge with a positive X value that it shares with the cube below it. This should look like an arm bending. Each cube will repeatedly rotate from 0° to 45° and back. At full rotation the top cube will be at the same level as the base cube. The five cubes will (roughly) form an arch. See figure 1.
- Implement simple view frustum culling. In the C / C++ code,
  - Calculate a bounding sphere for each box. Transform the center of the bounding sphere by the model-view matrix.
  - Calculate the plane equations for the camera-space view volume. This is probably the most difficult step.
  - Using the method described in the lecture notes to determine whether or not a sphere is inside the view volume.
  - Do not render cubes associated with spheres that are outside the view volume.
  - To test this, perform culling for a view volume that is much smaller then camera's actual view volume. Using half the actual field-of-view is a good choice. In addition, it is useful in this mode to have a hot key to enable or disable the frustum culling.

For extra credit, draw all five cubes using a single call to `glDrawElementsInstanced`. The five transformation matrices will be calculated in advance and passed to the vertex shader as an array (i.e., `mat4 mvp[5];`). The vertex shader built-in variable `gl_InstanceID` will be used to select the correct transformation for each instance.

4 Lighting - due 16-November-2011

The final part of the assignment is to add lighting to the scene. The majority of the code for this portion of the assignment will be in GLSL.

- Supply per-vertex normals
  - In addition to per-vertex position and color, specify per-vertex normals.
  - Create a new `attribute` in the vertex shader called `normal` and pass the per-vertex normals in through this attribute.
  - In addition to passing in the model-view-projection matrix, pass the upper 3x3 portion of the model matrix. Call this new matrix `normal_transform` in the vertex shader.
  - Transform the vertex normal by `normal_transform`. Question to think about: what "space" is the transformed normal in?
- Modify the vertex shader to perform per-vertex lighting.
  - Supply the position of a point light to the vertex shader in a uniform called `light_pos`. The point light should orbit the cubes around the (world-space) Z-axis. The point light should be 8 units from the origin.
  - Supply the direction of a directional light to the vertex shader in a uniform called `light_dir`.
  - Calculate the diffuse and specular lighting contributions for the point light.
  - Calculate the diffuse and specular lighting contributions for the directional light.
  - Combine the lighting from both lights with the vertex color. Pass the resulting color to the fragment shader in a varying called `color`.

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).