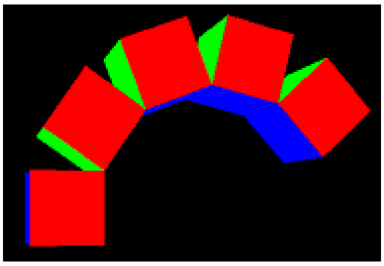

Figure 1: Arch of rotating cubes

CG Programming I – Assignment #3 (Cube arch scene)

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

1 Support Routines - due at the end of class 17-October-2012

In the first part, you will implement a series of C/C++ routines that will form the basis of the remaining parts.

- 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. *Note:* This part is not due until the end of class 24-October-2012.

  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.) or look-at functions.

  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`, etc. These functions should *always* be called from `main` as early as possible. This helps identify regressions quickly!

  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 Cube Arch - due at the start of class 24-October-2012

The second part actually does some rendering!

- Generate a series of transformation matrices for a set of five cubes. 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.

- Using the `look_at` function, have the camera slowly orbit the stack of cubes.

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

3 Instanced Drawing - due at the end of class 24-October-2012

Modify the drawing code (`main.cpp`) and add new shaders so that all five cubes can be drawn in a single draw call. The five transformation matrices will be calculated in advance and passed to the vertex shader as an array (i.e., `mat4 mvp[5];`). The built-in variable `gl_InstanceID` will be used to select the correct transformation for each instance. Drawing is performed using `glDrawElementsInstanced`.

Allow the user the ability to toggle between instanded and non-instanced drawing using the 'i' key. Since performance data is logged while the test is running, make not of the change, if any, between the instanced and non-instanced versions.

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).