CS488/688

Fall 2017

Assignment Specifications

University of Waterloo

Department of Computer Science

Instructor(s): Stephen Mann

August 31, 2017

Contents

Chapter 1

CS488/688 F17 Assignment Format

"I take off marks for anything..."

– A CS488 TA

Assignments are due at the beginning of lecture on the due date specified. More precisely, all the files in your assignment directory *must* be dated before the beginning of lecture, and the documentation *must* be handed at the beginning of class.

The documentation for the assignment will be submitted in class. In addition, we will examine the files under your `cs488` subdirectory. The portion of the assignment that is physically handed in must be arranged in the assignment format documented below. Failure to follow any of the procedures specified here will result in deductions from your assignment mark.

There is a checklist available at the end of this section. It is *strongly* suggested that you make a few photocopies of this checklist, and step through it carefully while preparing your submission. However, you should *not* hand in your checklist with your submission.

Invalid Excuses for Late Assignments

The following excuses will **NOT** be accepted for late assignments:

"The computer was down."
The computer often goes down shortly before assignments are due. Take this into consideration when working on your assignment.

"The print queue was too long."
The print queue is *always* too long shortly before assignments are due. Print your documentation early.

"I ran out of money on my print account."
It doesn't cost much to print the assignments. Just make sure you have enough money in your account.

Assignment Specification Format

Each assignment specification has the following sections:

Topics: A list of topics covered in the assignment

Statement: A statement of what you need to do for the assignment. This may be followed by additional sections that give more detail on particular topics.

Donated Code: This is code that we give you to get you started on the assignment and/or some routines that we supply for your use. Note that we will only list `.cpp` files here; however, there are often associated `.h` files.

Deliverables: This is a list of files that you must submit (in addition to what's listed in the next section) and details what you should be sure to include in your documentation, plus anything else you need to submit with an assignment.

Objectives: This is a list of objectives on which your assignment will be graded. Each objective has equal weight in your grade. You should submit a copy of this page with your written documentation. *Be sure to sign this sheet at the bottom!*

Do not check off objectives on the sheet you submit—the TA's fill in the blanks next to each objective when determining the mark for your assignment.

Directory Structure

The TA(s) will execute your program and look at the source code in your `cs488` subdirectory. Special group ownership and permissions are necessary for us to grade your assignment. We will change the permissions when we grade, so it is important that your directory is set up correctly by the `setup` programs in `/u/gr/cs488/bin`. If you register for the course late, it may take a while for MFCF to get your account established. Until your account is set up, we cannot grade your assignments.

Note that the `grsubmit` program sets file permissions to allow the TAs access to your files. It also generates a checksum. You must submit a hardcopy of this checksum (the output of `grsubmit`) with your documentation. No files are transmitted to us via `grsubmit`; you can run it multiple times with no problems. However, after running the `grsubmit` program for the final time **DO NOT MAKE CHANGES TO THE FILES OR DIRECTORIES, OR TO THEIR PERMISSIONS.**

Note to grad students: You will not have an account on the SGI workstations until you register, so it is important that you register for the course as soon as possible. Once you register, you will get a `cs688` directory in the *cs688* group. The `setup` command will create a symbolic link between `cs488` and `cs688`.

If you are not yet registered, you should first make the `cs688` directory yourself. Eg, type `mkdir ~/cs688` before running `setup`.

When you have finished working on an assignment, you should remove all unnecessary files (i.e., core files, temporary files, `.o` files, etc). Only the Python scripts and C++ source code, the dynamic libraries, and any input or output should remain in the directory.

For some assignments, the assignment specification (under the Deliverables section) will name certain files in which to place your code, etc. You must use these names no matter how much you may disapprove of them. In addition to what is listed in each assignment specification, the `cs488/handin/A`*n* directory should contain the following entries (and nothing else):

1. The executable(s) required for the assignment, *named as indicated in the assignment handout.* Regardless of how much you may disapprove of these names, you are expected to use them.

2. A subdirectory `handin/An/src`, which should contain all source files.

3. Your mainline Python script(s). Subroutines that are imported should be under the `src` or `data` subdirectories.

4. A file called `README`. This file should indicate on which machine each executable(s) was compiled and linked, should explain *briefly* how to run each program, should explain what assumptions you made (if any), *and should contain nothing else.*

5. A subdirectory `handin/An/data`, which should contain any data files, and subdirectories other than those specified in the assignment that were used to test your program. If you have gone to the trouble of creating some interesting data files of your own, make reference to them in the `README` file.

6. One or more files called `screenshotXX.png` that are screenshots of your program. You must have at least one screenshot file called `screenshot01.png`. You may have additional screenshots, which should be number consequetively following 01. For most assignments, you will likely only want a single screenshot. However, for the project, you may wish to have more screenshots. For the raytracer (assignment 4), your `screenshot01.png` file should be your best image. In general, you should take a screenshot that reasonbly illustrates that your program works.

The README file and all source files (e.g., `.cpp`, `.h`, and `.py` files) should contain your name, userid and student number near the top of each file.

DO NOT REMOVE OR MODIFY ANY OF THE ABOVE FILES UNTIL WE HAVE RETURNED YOUR GRADED DOCUMENTATION! The TA(s) will look at these files and check that the modification dates match those given by the checksum you submit with the documentation.

Documentation Submission

You will also be required to submit some documentation. This documentation should include the items listed in the following subsections. The documentation should be ordered in the same sequence as the following subsections, and all pages should be stapled together. **MARKS WILL BE DEDUCTED IF THE DOCUMENTATION IS NOT STAPLED TOGETHER.** Paperclips are **not** acceptable.

Title Page

The first page should be the title page. The title page should list your name, your userid, your student number, and the assignment's number. Leave blank space for the TAs' comments.

Objective List

Each assignment specification includes an **objective list** as the last page. You should include a copy of this sheet as the second page of the document you submit. The objective list indicates how the assignment is to be marked, and serves as a marking aid for the TA(s). Usually, there will be ten objectives worth one mark each. If you don't have time to complete the entire assignment, try to achieve as many of the objectives as you can in the time you have.

Be sure to fill in the information at the top of the Objective sheet, even though this information should also be on the title page, and make sure you read and sign the declaration at the bottom of the page. *No credit will be given for the assignment if you have not signed the Objectives sheet.*

Manual

This should NOT be a restatement of the assignment specification. You will be given strict procedures to follow on how to begin execution, what data is used and what interaction with the program is required. It is not necessary to reiterate this in great detail.

The manual should include any details or assumptions that you have made that were not specified in the specification. These could include additional commands, options or features or any additional data files that you may have generated. If you have made any assumptions about the assignment specification or the objectives, be sure you state and justify them.

You should also note in your manual any objectives that you did not complete. If you wrote code for objectives that you did not get completed, you should request code credit in your manual. See the section on code credit for details on what to put here.

Hardcopy

Hardcopy refers to any paper documentation requested, such as PostScript output or other diagrams.

Checksum

You will need to submit a hardcopy of the output of the program `/u/gr/cs488/bin/grsubmit`. To run this program, change directory to your `cs488/handin` directory and run `/u/gr/cs488/bin/grsubmit`. with the current assignment as an argument. For example, for Assignment 1 you would type

```
grsubmit A1
```

(assuming `/u/gr/cs488/bin` is on your path) and submit the resulting output. Again, **DO NOT MODIFY ANY OF YOUR FILES FOR THE ASSIGNMENT ONCE YOU HAVE RUN THIS PROGRAM!**

Program Execution

Each assignment handout specifies the names of the executable program(s) and scripts to be generated, and the names of supplied data files and scripts provided under `/u/gr/cs488`.

Follow the instructions in the assignment specification for naming conventions for data files and access. If the assignment specification says that the standard input `stdin` or a filename specified on the command line will be used, make sure to implement this functionality. This is because the TA's may use additional test scripts for marking.

Similarly, any textual output required should be sent to the standard output (`stdout`) or to a filename provided on the command line or though a graphical user interface, as specified. Output should not be sent to a file whose name has been hard coded into the executable.

Your final submission should NOT send debugging output to `stdout` or `stderr`. Debugging output should be turned off for the submitted versions.

README vs Manual

You will notice that you need to prepare both a `README` and a Manual for each assignment. In essence, the `README` tells us how to run your program, while the manual tells us what you did and why you chose to do it that way. Thus, the `README` should contain the following information:

- Which machines the executable(s) were compiled on;
- How to run your program, including the following information:
  - How to invoke your program.
  - How to use the parts of your user interface that are not specified in the assignment description.
  - Any assumptions about how the user will interact with your program that, if violated, might cause your program to fail.

Note that the `README` does not have to be handed in as hardcopy. We will read it online. You will find a skeleton README file in `/u/gr/cs488/data/README.skel`.

The manual should contain justification for your decisions and extra information about the implementation, including:

- Justification of any choice you make, whether in choosing your user interface, selecting a data structure, etc.
- Command-line options.
- Extra features or data files.
- Mention any objectives you did not complete, and (possibly) request code credit.

CS488 Assignment Checklist

Paper submission has:

[__] Title page with name, student ID number, course (CS488 or 688), assignment number, and userid. If there are multiple sections of the course in the semester, be sure to include the name of your professor and/or the section number as well.

[__] Second page is an objective list:

- Fill in the information at the top
- SIGN IT!
- DON'T check off objectives yourself

[__] Third page is a MANUAL:

- Describe anything unusual about your particular assignment.
- Ask for code credit.
- Indicate what doesn't work.
- Indicate any "interpretations" of spec.
- Don't restate the assignment specification.
- Be concise.

[__] Last page is a checksum printout from `grsubmit`.

[__] All of pages are STAPLED together in the top left.

Code has:

[__] Executable and mainline scripts are in:

```
~userid/cs488/handin/A?/*
```

[__] All filenames are as per spec.

[__] Running the executable or mainline script from this directory works and produces NO DEBUGGING OUTPUT

[__] README file indicating UI details, "how to run", is in:

```
~userid/cs488/handin/A?/README
```

[__] Screenshot file(s) of your program, is in:

```
~userid/cs488/handin/A?/screenshot01.png
```

(additional screenshots maybe provided if desired).

[__] C source code and Python scripts are in:

```
~userid/cs488/handin/A?/src
```

[__] Data files (i.e., scenes) are in:

```
~userid/cs488/handin/A?/data
```

[__] Running the executable conforms to objectives 1-10 unless otherwise indicated in the MANUAL. Always re-read the spec WHILE running the final program to make sure that all details are correct.

The most common mistake is to forget to sign the objective list. By checking this list before handing in everything, you can avoid losing marks for these nit-picky details.

Chapter 2

CS488/688 F17 A0: Introduction

"I want to work on Assignment 0. I want to do some Tcl programming—there's no thinking involved!"

– The TA who developed the Tcl version of this assignment

This assignment is due **Thuesday, September 19th [Week 2]**.

2.1 Topics

- Familiarise you with the course computing environment.
- Ensure that your account(s) are set up properly.
- Provide exposure to Qt and a refresher of C++.
- Provide practice in building/modifying a user interface.
- Give you a trial run at submitting an assignment.

2.2 Summary

This assignment is optional.

If you do not hand it in, you will not lose marks. If you do hand it in, it will be marked in the normal way, but the mark will not be recorded. Therefore, if you have any theologically unsound habits that will cost you marks or if your account is not set up correctly, you will have a chance to correct things before Assignment 1.

In past terms, over half the students who did not submit Assignment 0 lost marks on Assignment 1 for setting up their account incorrectly or for submitting inadequate documentation. These mistakes could have been avoided had these students completed Assignment 0.

It is in your best interest to do this assignment. You will gain some information and experience which will be useful for later assignments. Thus, even if you do not choose to hand in Assignment 0, you will still eventually need to learn what it covers before you can finish later assignments.

Note also that the directions in this assignment are more explicit than in the remaining assignments. In the future, the steps to follow will not be written as explicitly.

2.3 Statement

Do the following:

1. Set up your account. To do this, login to a Linux or Unix machine in the undergrad environment and type "`/u/gr/cs488/bin/setup`". Observe that you now have the following directory subtree in your home directory:

```
cs488/
    handin/
        A0/
            src/
            data/
        A1/
            src/
            data/
        A2/
            src/
            data/
        A3/
            src/
            data/
        A4/
            src/
            data/
        A5/
            src/
            data/
```

2. The next steps assume the that `qmake` and `qt libraries` are installed. These should already be installed on the machines in the graphics lab. If you are working on your own machine, and you don't have `qmake` and `qt` libraries installed, run the following script:

```
sudo apt-get install qt5-qmake qt5-default libqt5opengl5-dev
```

3. You will find several `.cpp` and `.hpp` files in the `A0` subdirectory, as well as a `paint488.pro` file. Enter the subdirectory and type `qmake paint488.pro`. This generates a `Makefile`. Then, type `make`. The program should compile and you should be left with an executable called `paint488`.

   The program is a simple painting program that allows you to draw rectangles, ovals and lines.

4. To run the program type `./paint488` at the command line. To exit you must select the "Quit" entry under the `Application` menu.

Modify `paintwindow.hpp` and `paintwindow.cpp` so that a "Quit" button is placed at the bottom of the window. When this button is pressed, the program terminates.

5. Add a Clear entry to the `Application` menu. This entry should clear the screen.

6. Add a new menu for selecting a colour. This menu should be labeled `Colour`, and should appear between the `Tools` and `Help` menus on the menu bar. You should be able to select the following colours: Black, Red, Green, Blue. Any subsequent draw of a rectangle or oval should fill the object with the selected colour.

   The default fill colour should be Black and the default shape should be Line.

   If you like, you can add additional fill colours, such as Cyan, Magenta, Yellow, Purple, Orange, White, but you will only be graded on Black, Red, Green, and Blue.

7. Change the Tools menu to use radiobuttons to select between the drawing primitives. See the Qt Documentation on `QActionGroup`.

8. Add Help entries to Help menu for Rectangles and Ovals.

9. Add the following keyboard shortcuts:

   C: Clear

   L: Line

   R: Rectangle

   O: Ovals

   Note that lower case letters should trigger the shortcuts. We have already implemented a keyboard shortcut for Quit; you should add your keyboard shortcuts to the same handler.

10. Write a Manual and a `README` for `paint488` explaining how to run and use it. The Manual should be submitted in hardcopy. The `README` should be in the `cs488/handin/A0` directory but does not have to be handed in as a hardcopy.

11. Remove all unnecessary files (eg., `*.bak, *~, core`, etc.).

12. From your `cs488/handin` directory run "`grsubmit A0`" to create a checksum for your assignment. Note that `/u/gr/cs488/bin` should already be in your search path. You are to print out this checksum and hand it in with your assignment.

13. Hand in all the things mentioned in the **Assignment Format** documentation that are applicable to this assignment. Double check the objectives to ensure that your assignment meets all of them. Make sure you sign the objective sheet and fill in the requested information!

2.4 Extras

If you wish to add some extras (sorry, no bonus points) to the paint program, consider adding the following:

- Rubberbanding. When drawing an oval or a rectangle, it is useful to draw the outline of the shape (or even the shape itself) as you move the mouse after the mouse button down.
- Additional shapes. Some additional shapes can be drawn if you allow multiple mouse presses. For example, you could draw a polyline (a sequence of line segments), an arbitrary polygon, or Bézier curves.
- Updating/deleting elements. You might want to change the colour of one of the elements you've drawn. Or you might want to delete one of the elements. This would require find the element and either removing it from the m_shape list or modifying its colour.
- Save/print. You could add additional functionality to either save the primitives to a file (to be read in later) or to print the screen.

Note that except for the first one, each of these will likely require a significant amount of coding.

2.5 Donated Code

In `/u/gr/cs488/data/A0` you will find

- `main.cpp` - The initial startup function `main`.
- `paintcanvas.hpp`, `paintcanvas.cpp` - The canvas onto which the user can draw
- `paintwindow.hpp`, `paintwindow.cpp` - The main window of the application.
- `paint488.pro` – Used to create a Makefile that includes all Qt libraries
- `Makefile` - Used to compile your program with `make`.

These files have been copied to your `handin/A0` directory.

2.6 Deliverables

Executables:

`paint488`. Be sure to place this in the correct directory (under `A0`).

Source code:

All source code should be placed in `A0/src`. You should not have to add any new files, but can if you wish. You should not have to change the Makefile. If you wish to add a file to the project, add your files in the `.pro` file in the `A0/src` folder.

Documentation:

`README`. Be sure to place this in the correct directory (under `A0`).

2.7 Objectives: Assignment 0

Due: Thuesday, September 19th [Week 2].

Name: _____

UserID: _____

Student ID: _____

___ **1:** The account is set up correctly.

___ **2:** The checksum program was run, and its output was printed and included in the submitted documentation.

___ **3:** The correct information was handed in as hardcopy, a `README` file exists in `handin/A0`, and the executable is in `handin/A0`.

___ **4:** There is a "Quit" button at the bottom of the window that terminates the program.

___ **5:** A Clear entry has been added to the Application menu, which clears the screen.

___ **6:** A Colour menu with radiobuttons has been added with the listed colours. After a colour is selected, any subsequent draws of a rectangle or oval are filled with that colour, and that colour's radiobutton is turned on.

___ **7:** The tools menu has been modified to use radiobuttons to select between the graphics primitives.

___ **8:** Keyboard shortcuts have been added for Clear (C), Lines (L), Rectangles (R), and Ovals (O). Both upper and lower case letters should trigger the shortcut.

___ **9:** Help entries for Rectangles and Ovals have been added.

___ **10:** A screenshot of your program is provided in `handin/A0/screenshot01.png`.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Chapter 3

CS488/688 F17 A1: OpenGL

"I think this assignment should have 8 objectives but we should still mark it out of 10."
– The Mean TA.

This assignment is due **Thursday, September 28th [Week 3]**.

3.1 Topics

- Exposure to OpenGL
- Callback-based program structure
- `Qt` user interfaces

3.2 Statement

This assignment will get you started writing graphics applications using OpenGL. It will also familiarize you with the set of languages and APIs we will be using for subsequent assignments. You will be writing a user interface in C++ using the `Qt` toolkit. The user interface will be wrapped around a window in which graphics will be rendered using OpenGL from the C++ application.

In particular, the program is a game in which the user must manipulate falling tetrominoes so as to make complete lines at the bottom of a well. When lines are completed, they are removed from the game. When the well fills up, the game is over. Any resemblance between this game and a popular arcade game of Russian extraction from the 1980s is purely coincidental.

You will also implement some graphical functionality not directly related to game play, but that will help you develop OpenGL 3.2 skills needed in later assignments.

3.3 The game

The game takes place in a U-shaped well of unit cubes enclosing a grid of width 10 and height 20 in which tetrominoes can fall. The blocks occupy discrete positions in the grid (they don't fall smoothly, but jump from position to position).

Tetrominoes start in a four unit tall stripe on top of the well. Every time a predetermined interval elapses, the current tetromino falls one unit. A value of 500ms is a good novice interval; 100ms is more challenging. At any time, the current piece can be moved to the left or right, rotated clockwise or counter-clockwise, and dropped the rest of the way down the well. When the piece can fall no further, it stops, and any rows in the well that are completely filled are removed from the game. The game ends when a piece cannot clear the starting stripe.

You should have at least three different speeds at which the pieces fall.

The game should be drawn from unit cubes. The well and the various piece shapes should all be drawn in different colours.

You should add a new game piece. The piece must not be a rotation or reflection or shift of an existing piece. However, your new piece does not need to be a tetrominoe; i.e., it may be composed of more (or less) than four cubes.

3.4 The interface

The user interface should be written in `Qt`, a cross-platform application and UI framework for C++ developers. You will need to implement the following functionality (the letters in () indicate the keyboard shortcut; remember, both upper and lower case should work for the keyboard shortcut):

- An `Application` menu with the following menu items:
  - `New game (N)`: Start a new game. `N`.
  - `Reset (R)`: Reset the view of the game.
  - `Quit (Q)`: Exit the program. (This one should already be implemented; be sure not to break it.)
- A `Draw Mode` menu with the following menu items:
  - `Wire-frame (W)`: Draw the game in wire-frame mode.
  - `Face (F)`: Fill in the faces in the game. Each different piece shape should have its own uniform colour.
  - `Multicoloured (M)`: Similar to `Face` mode, but each cube has six faces of different colours (i.e., no two faces should have the same colour).

  The `Draw Mode` menu should use radio buttons to indicate which state is selected.
- A radiobutton `Speed` menu with at `Slow (1)`, `Medium (2)`, and `Fast (3)` speeds that sets the rate at which pieces fall. The game may use additional speeds, but you need to be able to set the speed to one of the three.
- Mouse movements:
  - Mouse operations should be initiated by pressing the appropriate mouse button and terminated by releasing that button. Only motion in the horizontal direction should be used.

  - The left mouse button should rotate the game around the X-axis.
  - The middle mouse button should rotate the game around the Y-axis.
  - The right mouse button should rotate the game around the Z-axis.
  - When the shift key is pressed, all mouse buttons should uniformly scale the game (both the board and the pieces). When the mouse moves to the left, the game should become smaller, and when the mouse moves to the right the game should become larger. The maximum and minimum scales should be restricted to a reasonable range.

  You must make reasonable decisions about how much to scale or rotate for every pixel's worth of mouse motion. For example, if the mouse isn't moving, there should be no scaling or rotation.

  You are also required to implement a feature sometimes known as "persistence" or "gravity". If, while rotating, the mouse is moving at the time that the button is released, the rotation should continue on its own. This decision should be made at the time of release; after that, it should persist independently of mouse movement, until the next button press.

- Game play:
  - The left arrow key should move the currently falling piece one space to the left.
  - The right arrow key should move the current piece one space to the right.
  - The up arrow key should rotate the current piece counter-clockwise.
  - The down arrow key should rotate the current piece clockwise.
  - The space bar should 'drop' the piece, sending it as far down in the well as it will go.

3.5 Qt and OpenGL

In this course, user interfaces are written in `Qt`, a cross-platform application and UI framework for C++ developers.

If you opt to not to use Qt, you will need to use the following OpenGL commands:

Shader Program	**Drawing Objects**	**Other**
glCreateShader	glGenVertexArrays	glEnable
glSourceShader	glBindVertexArray	glDisable
glCompileShader	glGenBuffers	glClear
glDeleteShader	glBindBuffer	glClearColor
glCreateProgram	glBufferData	
glAttachShader	glEnableVertexArray	
glLinkProgram	glVertexAttribPointer	
glDetachShader	glDrawArrays	
glDeleteProgram	glDisableVertexArray	
glValidateProgram	glVertexAttrib*	
glUseProgram	glUniform*	
glGetAttribLocation		
glGetUniformLocation		

(Of course, you may find that you will want to use additional OpenGL functionality to add extra features. Note that the `Matrix4x4` class stores its matrices in row-major order, but OpenGL expects matrices in column-major order.)

If you have chosen to use Qt, you will use the following objects which replace some OpenGL calls and classes you have to make.

For more information, go to Qt Documentation.

`Classes`

- QOpenGLBuffer (replaces gl(Gen—Bind)Buffer(data)) - Qt 5.1+
- QOpenGLArrayBuffer (replaces gl(Gen—Bind)VertexArray) - Qt 5.1+
- QGLBuffer (same as QOpenGLBuffer) - Qt 5.0
- QGLShaderProgram (replaces all Shader Program calls)

Each class has replaces certain OpenGL calls while making the interface simpler. Please look at the documentation for the full class descriptions.

3.6 The Skeleton Program

If your account is correctly set up, you will find a skeleton program in the `cs488/A1/src` subdirectory of the source distribution. The program creates a user interface with an OpenGL window. As a test, it draws triangles where the corners of your game (not including the well) should appear. The camera is set up so that the triangles appear centered and correctly sized. You need to modify this code to render the current state of the game and respond to user interface events. Here's a to-do list, with a suggested order that will help you make your way through the assignment.

- Write a function to draw a unit cube using OpenGL.

  If you're already familiar with OpenGL 3.2, you can write a single cube to a Vertex Buffer Object (VBO). You would then create a `Matrix4x4` model matrix for each cube and use the matrix functions to translate, rotate, scale this model.

- In your render function, draw a U-shaped border for the well out of cubes.

- Implement face rendering and wireframe rendering.

- Implement rotation and scaling. You should be able to see the effect on the well.

- A new piece shape has been added.

- Add code to draw the current contents of the game. Each piece type should be drawn in a different colour; the colours are up to you.

  Note that color manipulation happens in the fragment shader.

- Hook up a simple timer (using the `QTimer` class) that calls down to the game's `tick` method and re-renders. You should be able to see pieces falling.

- Implement the rest of the controls for game play and the remaining user interface details.

3.7 Donated code

The skeleton program comes with the following files:

- `main.cpp` – The entry point for the program.
- `viewer.hpp`, `viewer.cpp` – The OpenGL widget. All of the OpenGL-related code is here.
- `appwindow.hpp`, `appwindow.cpp` – The application window code. Most of the UI-related code (menubars, etc.) is in these two files.
- `game.cpp`, `game.hpp` – An engine that implements the core of the falling blocks game.
- `game488.pro` – Used to create a Makefile that includes all Qt libraries
- `Makefile` – Used to compile the program with `make`.
- `shader.vert` – Vertex Shader used to determine the position of every vertex
- `shader.frag` – Fragment Shader used to determine the color of each pixel

You should be able to get the skeleton program running using the commands `qmake game488.pro; make; ./game488`.

Additional information can be found at

- Qt Documentation: `http://qt-project.org/doc/qt-5/index.html`

3.8 Deliverables

These executable files should be put in the directory `cs488/handin/A1`:

- `game488` – The program executable.

All source files should be in the directory `cs488/handin/A1/src`.
Don't forget `screenshot01.png`

3.9 Hints, tips, and ideas

There are lots of ways this simple application could be modified to enhance playability and attractiveness. You are encouraged to experiment with the code to implement these sorts of changes, as long as you have already met the assignment's basic objectives. Here are some suggestions:

- As the game progresses (ie, as more filled rows are removed from the game), have the pieces fall at a faster rate.
- A scoring mechanism.
- Head-to-head networked play.
- Modified cubes for pieces. The blocks look much better if individual cubes have their edges slightly beveled.
- Animations for certain events. The board can spin around when you lose, for example.
- Add lighting.

If you make extensive modifications to the game, you should make sure to run in a "compatibility mode" mode by default – you should support at least the user interface required by the assignment. You can activate your extensions either with a special command line argument or a menu item.

3.10 Objectives: Assignment 1

Due: Thursday, September 28th [Week 3].

Name: _____

UserID: _____

Student ID: _____

___ **1:** Wireframe mode works.

___ **2:** Face colour mode works.

___ **3:** Multicoloured face mode works.

___ **4:** Pieces fall at three or more speeds.

___ **5:** A new piece has been added to the game.

___ **6:** The user interface works as specified (menus, mouse interaction, etc).

___ **7:** The game can be rotated.

___ **8:** The game can be scaled.

___ **9:** The game is playable (i.e., you can move the pieces as described under "game play" of the assignment specification).

___ **10:** Persistence works for rotation.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Chapter 4

CS488/688 F17 A2: Transformation Pipeline

"Can I use a dialog box to implement rotations?"
– A student working on Assignment 2 (who was promptly told 'No')

This assignment is due **Thursday, October 12th [Week 5]**.

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A2` from your CSCF account.

4.1 Topics

- Modeling and viewing transformations.
- Perspective and homogeneous coordinates.
- 3D line/plane clipping.
- Menus, menubars, valuators, and message areas.

4.2 Statement

You are to write a program that displays and interactively manipulates a wire-frame box that you should construct with vertices at the 8 unit points (±1,±1,±1).

You are to provide a set of coordinate axes (a **modelling gnomon**) that you should construct with three lines, drawn from (0,0,0) to (0.5,0,0), (0,0.5,0), and (0,0,0.5) respectively. This gnomon will represent the local **modelling coordinates** of the box, and it must be subjected to every modelling, viewing and projection transformation applied to the box *except scaling*.

You are also to draw a separate set of axes for the **world coordinates**, which are the same size (coordinates) as the modelling gnomon, but are aligned with the world coordinate axes instead of the modelling axes. The world gnomon should be at (0,0,0). You should subject this **world gnomon** only to the viewing and projection transformations.

Note that the gnomon are merely graphical representations of the coordinate axes; for the coordinate axes themselves you should use orthonormal bases.

You are to apply **modelling** transformations to the box (*rotations*, *translations*, and *scales*) and **viewing** transformations to the eyepoint (*rotations* and *translations*). Transformations will be menu-selected and will be applied according to mouse interactions. Specifically, x motion of the mouse will be used as a *valuator* to control the amount of each transformation, the mouse *buttons* will be used to select the axis of the transformation, and a menu will be used as a *choice* device to determine the major modes of the program's execution.

You will need to maintain four distinct coordinate systems in this assignment. Three of these coordinate systems are 3D and one is 2D: the box ("model") coordinates (3D), the eyepoint ("view") coordinates (3D), the universal ("world") coordinates (3D), and the display ("screen") 2D normalized device coordinates (which arise from the perspective projection of the eye's view onto the eye's x, y plane).

The modelling transformations apply with respect to the model coordinates (i.e. a *model mode rotation* about the the x axis will rotate the box about its current x axis, not the world's x axis). The viewing transformations apply with respect to the view coordinates (i.e. a *view mode rotation* about the x axis will appear to swing the objects of the view up or down on the screen, since the eye's x appears parallel to the screen's horizontal axis). None of the modelling transformations will change the world coordinates (i.e., the world gnomon never changes its location, though, of course, it may drift out of our view as a result of viewing transformations).

You are to form all matrices yourself and accumulate all matrix products in software. You will also do the perspective projection yourself, including the division to convert from 3D homogeneous coordinates to 2D Cartesian coordinates. This means that you will have to do a 3D near-plane line/plane clip in the viewing coordinate system to avoid dividing by zero or having line segments "wrap around".

4.3 The Interface

As with the previous assignments, this assignment is implemented in C++ with Qt. We have provided some code for you that sets up a window and draws an example set of lines for you. You will need to add the parts that do all the 3D transformations, projections, etc.

We suggest that you add a few matrices to `Viewer` (in `viewer.hpp`) – which ones you add and what each means is up to you. At the very least, you will need a modelling matrix and a viewing matrix.

You should implement the stub functions already found in `Viewer` and perhaps add some more of your own. You will also want to implement the missing matrix functions in `a2.hpp` and `a2.cpp`, and use these in your `Viewer`.

Your viewer should have an on-screen indication of where the near and far planes are located (from a call to `Viewer::set_perspective`). Simple textual labels are fine. Document how you display this in your manual.

4.4 Drawing lines

In `draw.cpp`, we provide the following C++ routines to draw lines and set colours in an OpenGL window:

- `draw_init(int width, int height)` — call this before drawing any line segments. It will clear the screen and set everything up for drawing.
- `draw_line(const Point2D& p, const Point2D& q)` — draw a line segment. `p` and `q` are in window coordinates.
- `set_colour(const Colour& col)` — set the colour of subsequent calls to `draw_line`.
- `draw_complete()` — call this after you are done drawing line segments.

These routines use OpenGL. Your assignment should not contain any further OpenGL calls. Further, you should not modify `draw.cpp`. You should call these routines from `viewer.cpp` with the GL widget active. There is already example code in `viewer.cpp` that does this for you.

4.5 Top Level Interaction

The menubar will support (at least) two menus: the `Application` and `Mode` menus. The `Application` menu will have two selections: `Reset` (keyboard shortcut `A`), which will restore the original state of all transforms and perspective parameters, and set the viewport to its initial state; and `Quit` (keyboard shortcut `Q`), which will terminate the program. The Quit menu item and shortcut should already be implemented in the provided code; be sure not to break it.

The `Mode` menu selections will be used to determine what effect mouse dragging on the viewing area will have on the transformations. The `Mode` menu will consist of a list of radiobuttons, which select among the viewing and modelling modes, and a viewport mode. There should be an on-screen indication of what mode is currently active (eg., a message bar).

In any View and Model interaction modes, transformations are initiated with the cursor in the 3D viewing area, upon a button down event. Relative motion of the cursor is tracked and the transformations are continuously updated until a button up event is received. The current interactive mode should be presented in a message bar somewhere on the display widget. You should also show the locations of the near and far plane.

If multiple mouse buttons are held down simultaneously, all relevant parameters should be updated in parallel. For rotation, you may apply the rotations in a fixed order, as opposed to composing multiple infinitesimal rotations. However, this ONLY applies to the case where multiple mouse buttons are held down; in general, you will want to be able to compose an *arbitrary* sequence of transformations.

These interaction modes are a bare minimum, and form a poor 3D user interface. We'll look at better ways to create an interface to 3D rotations in Assignment 3.

4.6 View Interaction Modes

The following view interaction modes should be supported:

Rotate (keyboard shortcut `O`): Use x-motion of the mouse to:

LMB: Rotate sight vector about eye's x (horizontal) axis.

MMB: Rotate sight vector about eye's y (vertical) axis.

RMB: Rotate sight vector about eye's z (straight into eye).

Translate (keyboard shortcut `N`): Use x-motion of the mouse to:

LMB: Translate eyepoint along eye's x axis.

MMB: Translate eyepoint along eye's y axis.

RMB: Translate eyepoint along eye's z axis.

Perspective (keyboard shortcut `P`): Use x-motion of the mouse to:

LMB: Change the FOV over the range of 5 to 160 degrees.

MMB: Translate the near plane along z.

RMB: Translate the far plane along z.

A good default value for the FOV (Field Of View) is 30 degrees.

4.7 Model Interaction Modes

The following model interaction modes should be supported:

Rotate (keyboard shortcut `R`): Use x-motion of the mouse to:

LMB: Rotate box about its local x axis.

MMB: Rotate box about its local y axis.

RMB: Rotate box about its local z axis.

Translate (keyboard shortcut `T`): Use x-motion of the mouse to:

LMB: Translate box along its local x axis.

MMB: Translate box along its local y axis.

RMB: Translate box along its local z axis.

Scale (keyboard shortcut `S`): Use x-motion of the mouse to:

LMB: Scale box along its local x axis.

MMB: Scale box along its local y axis.

RMB: Scale box along its local z axis.

The initial interaction mode should be model-rotate, and this mode should be restored on a reset.

The amount of translation, rotation, or scaling will be determined from the relative change in the cursor's x value referenced to the value read at the time the mouse button was last moved. Make sure your program doesn't get confused if more than one button is pressed at the same time;

all the motion events should be processed simultaneously, as specified above, although individual "incremental" transformations can be composed in a fixed order.

You should use appropriate scaling factors to map the relative mouse motion to reasonable changes in the model and viewing transformations. For example, you might map the width of the window to a rotation of 180 or 360 degrees.

Do not limit the *accumulation* of rotations and translations; i.e., there should be no restriction on the cumulative amount of rotation or translation applied to an object, or to the number of sequential transformations.

4.8 Viewport mode

The viewport mode allows the user to change the viewport. Assume that you have a square window into the world, and that this window is mapped to the (possibly non-square) viewport. The window-to-viewport mapping should be as described in the lecture notes and the course text: if the aspect ratio of the viewport doesn't match the aspect ratio of the window (i.e., the viewport is not square), then the objects appearing in the viewport will be stretched. Further, when you change the viewport, you will see the same objects in the new viewport (possibly scaled and stretched) that you saw in the old viewport.

You should draw the outline of the viewport so that we can tell where it is.

In the Viewport mode, the left mouse button is used to draw a new viewport. The left-mouse-button-down event sets one corner, while the left-mouse-button-up event sets the other corner. You should be able to draw a viewport by specifying the corners in any order. If a mouse up position is outside the window, clamp the edges of the viewport to the visible part of the window.

The initial viewport should fill 90% of the full window size, with a 5% margin around the top, bottom, left and right edges. This is important so that we can verify that your viewport clipping works correctly – if you do not do this, you may lose marks in two places. The user should be able to set the viewport to any portion of the window, including sizes large than the original size. Note also that the viewport is to be reset to the initial size when the reset option is selected from the file menu.

The keyboard shortcut for viewport mode should be `V`.

4.9 Projective Transformation

You will need to implement a projective transformation. This will make the cube look three-dimensional, with perspective foreshortening distinguishing front and back. You may use a projective transformation matrix, if you wish. However, note that for this assignment there is no need to transform the z-coordinate. You can use the mappings x/z and y/z, although note that some additional scaling will be necessary to account for the field-of-view.

4.10 Orthographic View

If you cannot get your projective transformation matrix working, you may implement an orthographic view (no perspective) instead. However, you will not get a mark for objective 7. You may

also want to implement an orthographic view first and do your projective transformations last.

4.11 Line clipping

You will need to clip your lines to the viewing volume. There are several ways to clip, any of which will suffice for this assignment. Note, however, that you *must* clip to the near plane before completing the perspective projection, or you will get odd behaviour and coredumps. You may find it easiest to clip to the remaining sides of the viewing volume after you complete the projection (since you will be clipping to a cube), but you may clip at any point in your program. Note that we will be testing clipping against all sides of the view volume.

4.12 Donated Code

In `/u/gr/cs488/data/A2`, you will find

- `draw.cpp` – The drawing routines.
- `appwindow.cpp` – A window to which you may add widgets.
- `viewer.cpp` – A widget that will display your rendering. This is where the core part of your code will go.
- `algebra.cpp` – Routines for geometry, lumpy toads, etc.
- `a2.cpp` – Matrix routines you should implement and use.
- `a2.pro` – Used to create a Makefile that includes all Qt libraries
- `Makefile` – Used to build your code with `make`

These files have been copied to your `handin/A2/src` directory.

4.13 Deliverables

Executables:

Your source should be in the directory `cs488/handin/A2/src`. Your executable should be in `cs488/handin/A2`.

Additional Documentation: Be sure to note the following in your documentation:

- How you set up the view volume clips, and what you called the function that implements these clips;
- What matrices you chose to store in `viewer.cpp` and their purpose.

4.14 Objectives: Assignment 2

Due: Thursday, October 12th [Week 5].

Name: _____

UserID: _____

Student ID: _____

___ **1:** All model transformations are carried out with respect to the box's local origin. (This means, for example, that an x translation will not necessarily be parallel to the world's x axis, if the box has been rotated about its y or z axis.)

___ **2:** Viewing transformations work as expected according to the eye's coordinates. This is indicated by where the world gnomon is displayed.

___ **3:** Model transformations are applied to the box gnomon, except that the box gnomon is carried along **unscaled**.

___ **4:** The transformations in all modes act smoothly **while** the mouse is being moved. Pressing two buttons at the same time results in the two transformations being performed together.

___ **5:** Rotations, translations, and scales can be invoked in any order. Interaction modes may be entered and left as often as desired. There are no restrictions that prevent model transformations from being applied after the view has changed, or view transformations after the box has been transformed. No matter what sequence of transformations is entered, the box never distorts so that its edges fail to meet at right angles (in 3D).

___ **6:** A menubar with pulldown menus is used, with the functionality specified in the assignment description, including a reset for all transformations, the use of radiobuttons, and on screen feedback indicating the current interaction mode, and near and far plane locations.

___ **7:** The perspective transformation has been correctly implemented, and the field-of-view can be changed as specified in the assignment description.

___ **8:** The viewport user interface and the viewport mapping works as specified in the assignment description, and the initial viewport is about centered with 90% maximum size.

___ **9:** Lines are clipped to the near and far planes. The near and far planes can be changed as specified in the assignment description.

___ **10:** Lines are clipped to the sides of the viewing volume.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Chapter 5

CS488/688 F17 A3: Hierarchical Modelling

"I consider this assignment to be a gift."
– A former CS 488 TA's evaluation of Assignment 3.

"NO IT'S NOT!!!"
– The response of the Fall 1996 CS 488/688 Class to the above comment

This assignment is due **Thursday, October 26th** [**Week 7**].

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A3` from your CSCF account.

5.1 Topics

- Hierarchical models and data structures.
- Matrix stacks, undo/redo stacks.
- Scene parsing/scripting with Lua.
- 3D Picking.
- Z Buffer and backfacing polygon hidden surface removal.
- Filled and lighted polygons.
- Display lists.
- Virtual trackball for 3D rotation.

5.2 Statement

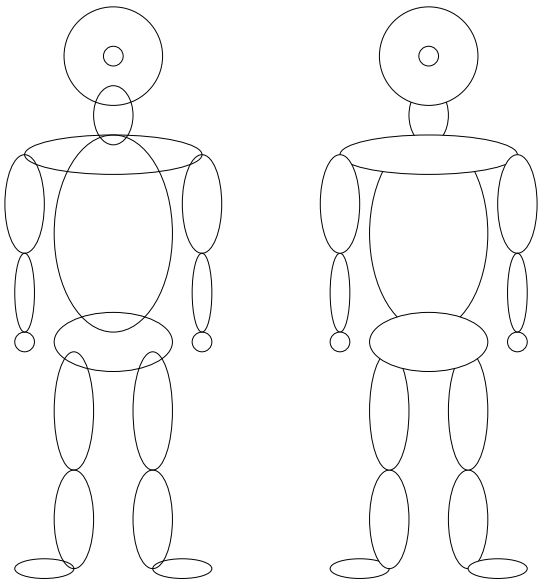
This assignment requires you to create a "puppet" in a hierarchical fashion from a number of instances of a transformed sphere primitive. You will build and manage a hierarchical data structure to represent the puppet. The puppet will be rendered and lighted interactively using OpenGL. You

will also build a user interface to selectively manipulate the joint angles of the puppet, and to globally rotate and translate the entire puppet. An undo/redo stack of joint transformations is maintained.

The faces of the sphere primitive should be composed of filled rectangles ("quads" in OpenGL) to make it appear solid, and should be drawn using an OpenGL display list for maximum efficiency. Models can be displayed with Z-buffer, backface polygon culling, and frontface polygon culling; each of these three OpenGL features can be turned on and off. The final puppet should be drawn using a suitable selection of lights and materials so that the 3D structure of the puppet is obvious. The position of the light sources should be fixed relative to the camera.

To create the puppet model, a sphere primitive should be appropriately transformed and instanced to create the torso (with its center as the origin of the puppet coordinate system). Smaller sphere instances for shoulders and hips should be positioned relative to this system. Their centres should form the origins for two secondary systems. With respect to the shoulder system, a neck sphere and two upper-arm spheres will form the origins of three subsidiary coordinate system origins. A sphere for the head will be positioned relative to the neck's center, and each sphere for the forearm will be positioned relative to the upper arm's center. The hands will be small spheres positioned relative to the forearms. The legs, consisting of thighs, calves, and feet, will be constructed similarly, with respect to the hip coordinate system.

The general effect should be as shown:

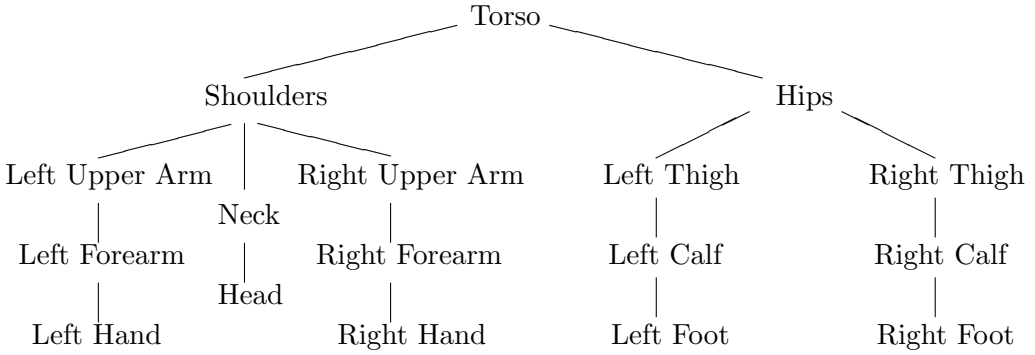


where the figure on the left shows all overlapping, scaled spheres, and the figure on the right has hidden surfaces removed (your puppet need not have "pigeon toes"). The circle in the center of the face is a nose. You need at least one feature on the head so we can determine if rotations of the head are correct; it doesn't have to be a nose. Feel free to add eyes, ears, hair, mouth, antennae, etc., to your puppet.

You can be creative with your model, as long as it has the same or more degrees of freedom (number of joints) as this one. In the past people have built models of gorillas, dogs, Teddy bears, aliens, dinosaurs, gerbils, Oktoberfesters, etc. However, a "creative" model is not part of this

assignment's requirements. Likewise you can add more modelling primitives if you want beyond the sphere, but it's not a requirement.

The coordinate system of each body part except the torso is therefore defined relative to a parent body part. The dependencies are as shown:



The nose has been left out of this hierarchy as it is part of the Head body part.

You are to implement primitives to build and render an arbitrary hierarchical model. You will then write a Lua script to build a hierarchical data structure to represent the puppet. Note the above structure is a physical representation of which part of the body is connected to what, not the actual data structure. For the actual data structure, you will need additional intermediate "transformation nodes" for each body part to get the spheres properly hinged at the ends and for the joints. It's also a good idea to not put children under nodes with nonuniform scales; create childless side branches instead. In particular, you will only want to put primitives at leaf nodes.

The definition of the puppet is to include **fifteen** free angles, three for each arm and leg, and three for the head and neck. These angles will be used to animate the puppet. The three arm angles will set the amount of rotation of the upper arm forward about the shoulder, the amount of rotation of the forearm forward about the elbow, and the amount of rotation of the hand backward and forward about the wrist, in the same plane as the elbow and shoulder.

The three leg angles will set the amount of rotation of the thigh forward about the hip, the amount of rotation of the calf downward about the knee, and the amount of rotation of the foot forward and backward about the ankle.

Of the remaining three angles, one will set the amount of rotation of the base of the neck forward with respect to the torso/shoulder, one will set the amount of rotation of the base of the head forward with respect to the neck, and the last will set the rotation of the head to the right or left relative to the neck.

All angles will have minimum and maximum values which the interface will enforce. For example, it should not be possible to rotate a knee or elbow the "wrong way", or to rotate a hand or foot more than 90 degrees from its neutral position.

Caution: In the past, students have found the creation of this model to be the most time consuming part of this assignment. You will probably want to design it on paper before writing the Lua code for the model.

5.3 Modelling

You are to model your puppet using a Lua script. The next section describes the Lua/C++ interface you need to implement. First, we take a quick look at how the interface is organized.

Lua is a simple scripting language. By using Lua to describe our scene we do not have to write a special-purpose parser for the scene. In C++, our scene will be represented as a number of `SceneNode` instances. Each `SceneNode` object has a transformation associated with it, and may have child nodes. There are two classes derived from `SceneNode`: `JointNode` and `GeometryNode`. These classes represent special types of nodes. Geometry nodes are nodes at which actual geometry is present (in this assignment, spheres). Joint nodes are nodes which can be manipulated in the user interface to rotate joints of the puppet.

To begin constructing a puppet, we need a root for our modelling hierarchy. We can get one by asking for a transform node and assigning the result to a Lua variable. The name passed to the function is useful for debugging purposes.

```
myroot = gr.node('root')
```

Functions like `gr.sphere` create new geometry nodes. We can then use `add_child` to make a newly-created sphere a child of the root node:

```
torso = gr.sphere('torso')
myroot:add_child(torso)
```

You may wonder why we used a colon instead of a period in the last line. In Lua, ":" is used to call member functions on objects and "." is used to call regular functions or class functions. We might then set the material properties of and transform the torso:

```
torso:set_material(gr.material(...))
torso:translate(1.0, 2.0, 3.0)
```

Finally, we simply return the root node of the puppet:

```
return myroot
```

Conceptually, the transformation at a node is applied to both its geometry and its children, and matrices deeper in the tree are premultiplied by matrices higher in the tree. This assumes that column vectors are used to represent points.

The tree given in the previous section just relates the pieces of the puppet. The tree you create for your program will need to have additional joint nodes, containing transformations and a joint rotation range. You will transform these "joint" nodes to animate your puppet. You may also need extra nodes to prevent child nodes from being affected by transformations meant to modify only the geometry of a parent node. Watch out, in particular, for scales. You generally will not want these in the middle of a chain of transformations.

5.4 The Interface

The interface of your program should have at least a menu bar and a viewing area. The menu bar will have (at least) an `Application` menu, an `Edit` menu, a `Mode` menu and an `Options` menu. In addition, when the graphics display is resized the aspect ratio should be properly maintained—the rendering of the puppet might change size, but its proportions should not become distorted.

The Lua interface functions have been written for you, but they call other functions which you will need to implement. Here is a list of Lua functions that have been implemented (in `scene_lua.cpp`):

- `gr.sphere(`*`name`*`)` — Return a sphere with name *name*. The sphere should be centered at the origin with radius 1.
- `gr.node(`*`name`*`)` — Return a node *name* that just contains a transformation matrix, which is initialized to the identity matrix.
- `gr.joint(`*`name`*`, {`*`xmin, xinit, xmax`*`}, {`*`ymin, yinit, ymax`*`})` — Create a joint node with minimum rotation angles *xmin* and *ymin*, maximum rotation angles *xmax* and *ymax* and initial rotation angles *xinit* and `yinit` about the x and y axes.
- `onode:add_child(`*`cnode`*`)` — Add *cnode* as a child of *pnode*.
- `gr.material({`*`dr, dg, db`*`}, {`*`sr, sg, sb`*`}, `*`p`*`)` — Return a material with diffuse reflection coefficients *dr*, *dg*, *db*, specular reflection coefficients *sr*, *sg*, *sb*, and Phong coefficient *p*.
- `node:set_material(`*`mat`*`)` — Give the node *node* material *mat*. Node materials can be changed at any time.
- `node:rotate(`*`axis, angle`*`)` — Rotate *node* about *axis* (`'x'`, `'y'` or `'z'`) by *angle* (in degrees).
- `node:translate(`*`dx, dy, dz`* — Translate *node* by *(dx, dy, dz)*.
- `node:scale(`*`sx, sy, sz`*`)` — Scale *node* by *(sx, sy, sz)*.

You can partially verify your implementation with `a3mark.lua` and `a3mark.png`. The TAs will use this script to test some of the functionality of your assignment.

Note that while the Lua-C++ interface for these commands is completely implemented for you (in `scene_lua.cpp`), you will need to fill in the stubs in `scene.cpp`, and extend the scene data structures appropriately.

5.4.1 Application Menu

The `Application` menu should have the following items:

`Reset Position` —Reset the origin of the puppet to its initial position. Keyboard shortcut `I`.

`Reset Orientation` —Reset the puppet to its initial orientation. Keyboard shortcut `O`.

`Reset Joints` —Reset all joint angles, and clear the undo/redo stack. Keyboard shortcut `N`.

`Reset All` —Reset the position, orientation, and joint angles of the puppet, and clear the undo/redo stack. Keyboard shortcut `A`.

`Quit` —Terminate the program. Keyboard shortcut `Q`.

5.4.2 Mode Menu

The `Mode` menu should have the following radiobutton selections:

`Position/Orientation` —Translate and rotate the entire puppet. Keyboard shortcut `P`.

`Joints` —Control joint angles. Keyboard shortcut `J`.

Initially, the program should be in the `Position/Orientation` mode.

The behaviour of this interface is detailed in the following.

`Position/Orientation:` The code in `/u/gr/cs488/demo/trackball` illustrates the interface you should implement for translating and rotating the puppet. You can use the `trackball.c` code here as a basis for your implementation.

The following details the behavior of each mouse button. If you have questions about how this interface works, test `intdemo` and make your interface match it.

- Dragging the mouse with LMB depressed changes the x and y translation of the puppet.
- MMB depressed changes the z translation; moving the mouse up on the screen should move the puppet away from the viewer, while moving the mouse down on the screen should move the puppet closer.
- Use RMB to implement a virtual trackball direct-manipulation-3D-orientation interface. The "virtual sphere" should be centered on the screen and its diameter should be 50% of the width or height of the raster widget, whichever is smaller.

  The skeleton code will display a circle centred on the screen. This circle corresponds to the virtual sphere. The radius of the circle will be a quarter of `min(screen width, screen height)`.

The translation should be done relative to the view frame. The rotation should also be performed relative to the view frame, translated to the puppet's origin. Since in this assignment the view frame is fixed, you can make this easy by putting the camera in a simple canonical world-frame position. Rotating the puppet about its own origin can be done with an appropriate transform node above the torso.

`Joints:` Clicking LMB is used as an event to signal picking. The cursor will be used with this signal to select a part of the puppet model.

Selecting is regarded as a toggle operation. If an item is unselected, and it is picked, it will become selected. If the same item is picked again, it will become unselected. Selecting an item should cause a visible change in the rendering of that item (change the material).

Only items that are directly under a movable joint should be selectable. When an item is selected, the joint immediately above it in the hierarchy will be affected by the user interface. If an item is not selectable, i.e. it is not immediately under a movable joint, picking it should do nothing. In particular, there should be no visible change in unselectable objects when they are picked.

Multiple simultaneous selections should be permitted.

For all selected items, the relative mouse y motion with MMB pressed will be mapped to the angles of the joint immediately above each selected object in the hierarchy. In addition, the head will rotate to the left and right with RMB depressed, in response to relative x motion of the mouse. It should be possible to depress both MMB and RMB simultaneously and simultaneously nod and rotate the head. The head should only move if it is selected, for both MMB and RMB motions.

5.4.3 Edit Menu

The `Edit` menu should have the following items:

`Undo` —undo the previous transformation on the undo/redo stack. Keyboard shortcut `U`.

`Redo` —redo the next transformation on the undo/redo stack. Keyboard shortcut `R`.

You should maintain an undo/redo stack of transformations. All joint transformations should be saved to the stack. The paradigm is that when you release a mouse button when in joint manipulation mode, the current joint angles are stored on the undo/redo stack. Initially, the stack contains the "reset" joint angles. If we think of the stack as extending upwards, you should maintain a current position in the stack. Selecting undo from the edit menu will restore the joint angles to the entry below the current one on the stack (and update the stack pointer to point at that entry). Selecting redo will update the joint angles to the entry above the current one on the stack (and update the stack pointer to point at that entry).

If several joint transformations are undone, and a new joint transformation is initiated, all undo/redo entries above the current one should be cleared. If Reset All or Reset Joints are selected from the File Menu, the undo/redo stack should be cleared. You should provide reasonable feedback to indicate that an attempt to undo or redo past the end of the stack has been attempted (and not allow the action).

5.4.4 Picking Menu

The `Picking` menu should be implemented if you are unable to get selection with the mouse working properly (objective 3). Create a checkbox menu item for each selectable part. You will not get a mark for objective 3, but you will be able to implement and demonstrate joint movements.

5.4.5 Options Menu

The `Options` menu should have the following checkbox item:

`Circle` —draws the circle for the trackball if checked. Keyboard shortcut `C`.

`Z-buffer` —draws the puppet with the OpenGL z-buffer if checked. Keyboard shortcut `Z`.

`Backface cull` —draws the puppet with backfacing polygons removed. Keyboard shortcut `B`.

`Frontface cull` —draws the puppet with frontfacing polygons removed. Keyboard shortcut `F`.

All options should default to being "off".

5.5 OpenGL

Most of what you need to learn about OpenGL is in the programming guide and man pages. However, here are a few items that are hard to find, and a few hints to make the assignment easier. None of the following are requirements on your program; they're just hints that students in past terms have found helpful.

- Unit normals. To get the lighting calculation correct, you must use unit normals. Although it is trivial to specify unit normals for the faces of the sphere, when you scale the sphere, the normals will also be scaled, making them of non-unit length. To avoid this problem, use the appropriate `glEnable` command to tell OpenGL automatically scale all normals to unit length.

- Material properties. You can set ambient, diffuse, and specular material properties. For debugging purposes, it is probably easiest if you turn off the specular material properties (or rather, if you never turn them on).

- Lights. For debugging purposes, it is easiest to have a single light source, placed at the eye. This will allow you to decide more easily if the shading of the puppet looks correct.

- Normals, Colours, Vertices. When you give the `glVertex` command, OpenGL assigns the normal specified by the most recently issued `glNormal` command; likewise for `glColor`, which can be set up to modify the diffuse reflection colour when lighting is enabled.

  Thus, when specifying the normals and colours of the vertices, you must specify them BEFORE you issue the `glVertex` command.

You are required to use a *display list* to render the sphere. A display list is way of storing all vertices and normals as they are calculated so that OpenGL can reuse them rather than have to recalculate (and retransmit) them to OpenGL. This is sometimes called "retained mode", as opposed to the usual way of using OpenGL, which is "immediate mode".

You must generate the sphere yourself using sines and cosines, and draw it as a set of triangles. You should select a resolution of the sphere that results in a good quality sphere, while still allowing for smooth motion.

5.6 Donated Code

In `/u/gr/cs488/data/A3` you will find

- `appwindow.cpp` – The application window.

- `algebra.cpp` – Routines for geometry, ferrets, etc.
- `main.cpp` – The program entry point.
- `material.cpp` – Classes and functions for manipulating materials.
- `primitive.cpp` – Classes and functions for manipulating geometric primitives such as spheres, etc.
- `scene.cpp` – Classes and functions defining modelling hierarchies. The layout of these classes should provide some ideas about how to organize your code.
- `scene_lua.cpp` – The Lua/C++ interface.
- `viewer.cpp` – The OpenGL viewer widget.
- `puppeteer.pro` – Used to create a Makefile that includes all Qt libraries
- `Makefile` – Used to build the program.
- `a3mark.lua` – A Lua script to test your implementation.
- `a3mark.png` – An image showing what the screen output of `a3mark.lua` should look like.

These files have been copied to your `handin/A3` directory. You should also use `trackball.c` which you can find in `/u/gr/cs488/demo/trackball`.

Note that the matrix multiplication routines are not needed for rendering, only to set up the matrices in the nodes.

5.7 Deliverables

Executables: The following two files are to be in your `A3/` directory:

`puppeteer` - Your puppet viewer. This program should take a single argument specifying a Lua puppet file to load. If no arguments are given it should load `puppet.lua` from the current directory.

`puppet.lua` - Your puppet.

The supporting source code should also be available in the `src` directory.

Documentation:

- Document changes you make to the data structure.
- Document the structure of the hierarchical model of your puppet.

For this assignment, you are encouraged to make creative models. The instructor and TAs may assign up to 1 bonus mark for interesting puppets.

5.8 Objectives: Assignment 3

Due: Thursday, October 26th [Week 7].

Name: _____

UserID: _____

Student ID: _____

___ **1:** `a3mark.lua` runs and displays correctly.

___ **2:** The puppet's proportions are reasonable and the spheres are joined together in a logical fashion. The spheres are "hinged" to their neighbours at their ends—they do not rotate about their centres.

___ **3:** Picking works correctly and reliably.

___ **4:** Selection records are kept correctly so that any sequence of picks-to-select and picks-to-unselect works. Multiple selections are supported. Selected portions of the puppet are clearly indicated—e.g. selected parts change colour, or their edges are highlighted.

___ **5:** The puppet can be globally rotated and translated for viewing purposes. The rotation user interface is a virtual trackball. A circle representing the virtual sphere can be turned on or off from the menu. It must be clearly visible when it is turned on. The puppet's configuration can be reset from the menu.

___ **6:** A well-designed hierarchical data structure is employed. Each sphere is drawn as an instance of a single display list sphere.

___ **7:** The joint movements are correct and the angles are restricted so that no grossly unnatural configurations are allowed. The puppet does not fly apart or distort during any sequences or combinations of actions.

___ **8:** Z-buffer, backfacing and frontfacing polygon hidden surface removal are implemented and each can be toggled on and off.

___ **9:** The puppet is lit such that its 3D structure is clearly visible.

___ **10:** An undo/redo stack is maintained.

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Chapter 6

CS488/688 F17 A4: Ray Tracing

> *"If you want to do a ray tracer for your project, you* **really** *need to complete Assignment 4 first."*

This assignment is due **Tuesday, November 11th [Week 9]**.

If you still need the provided code for this assignment run `/u/gr/cs488/bin/setup A4` from your CSCF account.

6.1 Topics

- Ray tracing.
- Inverse transformations and ray transformation.
- Ray intersection with spheres, cubes, and meshes of convex planar polygons.
- The Phong lighting model.

6.2 Statement

A ray tracer comprises a few major pieces of code to

1. Cast rays from the eye point through each pixel,
2. Intersect the rays with the objects in the scene,
3. Cast shadow rays from the point of intersection to each light source, and
4. Shade the rays, summing the contribution from each visible light source, and assign a colour to the pixel.

You are to write such a ray tracer. This ray tracer needs to support spheres, cubes, and meshes of convex, planar polygons as its basic geometric primitives. The ray tracer modeling language will

be implemented as a set of Lua callbacks, so scenes—including procedurally generated scenes—are specified as Lua scripts.

If you are planning to render a large scene (as opposed to a small test scene, small being less than 32x32 rays) please use the machines in :LabRoom:: raytracing takes a lot of sustained CPU time, which CFCF frowns upon for time-shared machines. On shared CFCF machines, if you render small test images, please "nice" your rendering jobs. Note also that many shells put a limit on how much CPU time a single program can use. To avoid running into this limit, run `unlimit cputime` before starting your ray tracer.

You are required to implement the following ray tracing functionality:

- Hierarchical transformations and models.
- The sphere, cube, and (convex, planar) polygonal mesh geometry types.
- Bounding volumes (sphere or boxes) for polygonal mesh types.
- Point light sources, with a Phong lighting model.
- Output of the resulting image to a PNG file.
- One additional feature of your own choosing, and a scene file (and resulting image) that demonstrates this feature.

Note: you may assume that all faces of a polygonal meshes are convex and planar. It is *suggested* that an intersection-of-halfspaces approach be used in 2D to test if an intersection point with the plane of a polygon lies within that (convex) polygon's area. Project the 3D plane/ray intersection point into 2D by dropping the coordinate corresponding to the largest value in the polygon's plane normal.

Use face normals for the cube and polygonal mesh model types to support shading. You do *not* have to support vertex normals. Note that Phong shading, i.e. normal interpolation, is *not* required—you don't have vertex normals anyway on the polygonal mesh type provided. However, you can implement Phong shading as your extra feature, as long as you can still read the existing test scripts. It is suggested that if you do this you add a new polygonal mesh primitive with a new Lua command, rather than modifying the interface to the one provided here. In general, implement the interface to your new features in a backward-compatible way.

You are also required to generate a test script named `sample.lua` that demonstrates the following: all of the primitive types required, the point light sources, at least one "shiny" surface (using the specular Phong reflectance model), and your additional features. This script should just write the resulting image to a `sample.png` file, and exit. You can look at the sample files in `/u/gr/cs488/data/A4` to get some ideas. You can also obtain some images corresponding to the test scripts from the course web page. These can be found in the A4 test scene link, under the gallery/exhibition section. When you submit the images rendered from these given scripts, please use the ORIGINAL scripts without any modifications. If you want to use these scripts to demonstrate your extra objective(s), such as transparency, reflectivity, etc., be sure to include the images generated from the unmodified scripts as part of you sample images.

You are only required to implement primary and shadow rays. Recursive secondary rays, needed to support reflections and transparency, are *not* required. Technical note: attenuation applies only to shadow rays, not primary rays!

You are also required to provide an interesting background that does not interfere with the objects in the scene. This means that colours should not be too bright, or the patterns too varied. Sunset effects could be tried, just as a suggestion. A constant background is not acceptable. You can parameterize the background either by pixel position or ray direction. The latter parameterization can be used, for instance, to make a sky background with dark blue at the zenith, white at the horizon, and brown below the horizon, or to implement a starfield for an outer space scene. Use your imagination. Your background should be in all your raytraced pictures.

The easiest way to implement a hierarchical raytracer is to transform each *ray* from the Viewing Coordinate System (VCS) to the Modeling Coordinate System (MCS) of each model. The transformed ray is then intersected with models in a canonical position, so you can take advantage of special forms of equations. To search the entire scene, for each ray you can do a recursive tree traversal, generating transformation matrices as you go. Alternatively, you can precalcuate all VCS-to-MCS transformations, flattening the tree; this will be somewhat faster (flattening cannot count as an extra objective, though). Keep in mind that in either case light sources should be specified in WCS, so you will have to transform the intersection from MCS back to WCS to do shading.

To compute the matrix used to transform the rays, you should extend the model callbacks to maintain an *inverse* transformation in each model node. Then compose these in reverse order as you descend the tree to find the necessary VCS to MCS transformation for each primitive. Object intersections then can be made more efficient (and simpler) by using a canonical form of an object in MCS.

6.3 Suggested Development

You are free to develop your code as you like. However, you will probably find it easiest if you first write a non-hierarchical raytracer (Objectives 1-6). Once you have that part of the code debugged, add the hierarchical part (Objectives 8,9; affine transformations, a general hierarchy, and bounding volumes for polyhedral objects). Depending on what you implement for your extra improvement, you may or may not want to complete Objective 10 before starting on hierarchical transformations.

Finally, you need to make a unique scene (Objective 7). While you can write a unique scene earlier in the development cycle, you may want to wait until you know if you will finish hierarchical models, since the power of hierarchical modelling will let you build a more interesting scene.

For this assignment, you are allowed to have some console output. For instance, you may want to output your render parameters and have some indication of how much progress your raytracer is making (i.e., 10%, 20% done etc). Printing out your entire hierarchy tree is probably too much output, though.

6.4 Cautions

This assignment is a lot of work. Although things look easy in the course notes, the details are a bit tricky to work out. What follows are a few hints from students who have worked in this assignment in the past.

- Numerical problems abound. In particular, watch for the following:
  - Try to minimise the number of times that you normalize vectors and normals. Each time you normalize, you introduce a small amount of error that can cause major problems.
  - The intersection of the ray and an object may actually be slightly inside the object. When casting secondary rays (such as shadow rays), the first object the secondary rays will hit will be the same object. To avoid this problem, discard all intersections that occur too close to the ray origin.
  - Use "epsilon" checks in your intersection routines, particularly the ray-intersect-polygon routine.
- In hierarchical ray tracing, on "the way down" you should transform the point and vector forming the ray with the *inverse transform.* On "the way back up" you should transform the intersection point by the transformation, and (assuming you represent the normal as a column vector) you should transform the normal with the *transpose of the inverse transform.* Potentially, this transformation of the normal will result in a non-zero 4th coordinate, in which case you should set the 4th coordinate to 0 (or, write a special matrix-vector product that ignores the fourth coordinate, and uses the transpose).

  To speed things up you way want to precompute and cache all the VCS-MCS transformations. Expand the DAG into a strict hierarchy first if you use caching.

6.5 Possible Raytracer Extensions

You are required to implement an "additional" non-trivial feature. There are many possibilities, such as an efficiency improvement or an addition of functionality. Several ideas are given below. They are purely a list of suggestions for your consideration; a list of papers on ray tracing is given in the course bibliography. These papers contain a lot more possibilities (some of them might even include code.)

6.5.1 Extra Functionality

Mirror reflections: This involves (recursively) issuing secondary reflection rays from the point of intersection. This would apply to objects that have been assigned a surface property with a non-zero "mirror coefficient".

Since purely reflective objects are rare, blend the colour found recursively with the colour used for shading a semi-reflective object using shadow rays, using the mirror coefficient. Note that you will have to specify a maximum recursion depth. If you use mirror coefficients $r < 1$, then the magnitude of r^n for the maximum r in the scene will give you an idea of how much

error will be made when truncating the recursion in scenes with multiple reflective objects. Even a recursion depth of 1 can generate interesting pictures, though.

Refraction: This involves generating (recursively) secondary refracted rays for objects that have a "transparency coefficient" and an index of refraction. Implementation is very similar to reflection rays. Use Snell's law to compute the direction of the refracted ray. [Whitted]

Aside: if you want to get fancy, the ratio between reflection and refraction is given by a function that gives more reflection at glancing angles. This is called the Fresnel effect and is a consequence of Maxwell's Laws; see the text or the references. For the purposes of this assignment, you can use constant coefficients for the amount of reflection and refraction. However, watch out for total internal reflection.

Supersampling: This involves generating multiple rays for each pixel and using some averaging function to combine the colours returned (e.g. sample on a 3×3 grid over the area of the pixel and average the nine colour values that are returned). This helps to reduce the "jaggies" and is the most basic form of **antialiasing**.

Other antialiasing methods: Generate more rays only where the scene changes (adaptive sampling), use random (stochastic) sampling techniques (jitter the sample positions in the subpixel grid), etc. There are many, many antialiasing techniques. [Cook : Stochastic] [Dippe : Stochastic] [Lee Uselton] [Mitchell] [Painter : Adaptive-Progressive Refinement]

Note: If you choose to implement a supersampling objective, make sure you include at least two comparison images of the same scene, one with supersampling turned on and the other without.

Fisheye/Omnimax Projection: Ray trace a 180 degree view, using a hemispherical "screen".

Spherical Lens Systems: Simulate a real lens system; use stochastic sampling across the aperture to simulate depth of field, and refractive intersection with sphere surfaces to create a lens system. (Even one lens is interesting; more would make a good project).

Additional primitives: Extend the modelling language and add primitives for (truncated) cylinders and cones. Note that these primitives are basically particular quadrics intersected with a pair of halfspaces, and (truncated) paraboloids and hyperbolids are in the same category. Quadric-based primitives are very easy to implement, especially since you have already been provided with a stable quadratic solver.

There are other possible primitive objects, such as superquadrics or tori, although these are harder than quadric-based solvers. For instance, a torus is generated by a quartic equation, which can be solved analytically, but it's hard to write a numerically stable quartic solver. Some kind of numerical root-solver is often required; reguli-falsi is recommended, but isolate the roots first using a geometric approach. [Many available references. Take a look.]

Texture-mapping: Determine the diffuse reflectance of objects based on stochastic or deterministic functions. This requires a function that computes, for each intersection point on the surface of a primitive, a set of texture coordinates. For instance, for a sphere, you can use the

elevation and azimuth of the intersection point to index the texture image. Alternatively, you can use a projective transformation of the MCS coordinates of intersection point. Computing the diffuse reflectance procedurally as a function of MCS spatial position (x, y, z) can be used to generating solid textures, like wood grain or marble. [Perlin,Hart]

See `http://textures.guinet.com/` for some sample textures.

Bump-mapping: Involves techniques similar to texture-mapping, but the texture functions perturb the object's surface normal, rather than diffuse reflectance, at a given point. [Blinn]

Lighting Models: Implement a decent "physical" lighting model, or some other alternative lighting model (like the Blinn-Phong model). Implement Phong shading (interpolation of vertex normals).

CSG: Extend the raytracer to provide CSG operations at model nodes (intersect, union, diff, etc) and extend the intersection routines to return 1D intersection intervals along the ray rather than a single intersection distance. Then at union nodes merge lists of intersection intervals passed back from the children, at intersection nodes compute the 1D intersection the intervals passed up from the children, etc. These computations can be performed using a simple count-up-at-entry and count-down-at-exit algorithm. Once the final set of intervals has been computed, take the first point of the first interval as the intersection point.

6.5.2 Improved Efficiency

The key to improving efficiency is the avoidance of unnecessary work, or rather, only applying work were it will make a difference. Some ideas for improving efficiency are:

Intensity thresholds: Check if the accumulated product of mirror reflectances is less than epsilon, then return black without checking for further ray hits (obviously goes along with doing secondary rays for mirror reflection).

Spatial partitioning: Modify your intersection routine to use a space partitioning scheme; e.g. BSP trees, uniform spatial subdivision, or octrees. Uniform spatial subdivision is particularly easy to implement and performs well in practice; a hierarchical version can be used as an extension of this in a ray-tracing project.

Note: simply flattening the DAG and caching transformation matrices, as suggested earlier, is *not* enough to get you credit for this objective.

6.6 The Interface

We have provided a set of stubbed Lua callback functions in C++. They implement a superset of the modeling language used in Assignment 3; you are expected to extend your code from that assignment.

What we list here are the additional functionality you need to add to the Assignment 3 language to get full credit in Assignment 4. You'll find that the Assignment 3 source code already provides stubs for this functionality.

- `gr.nh_box(`*`name, (x, y, z), r`*`)` — Return a non-hierarchical box with name *name*. The box should be aligned with the axes of its MCS, with one corner at (x, y, z) and the diagonally opposite corner at $(x + r, y + r, z + r)$.

- `gr.nh_sphere(`*`name, (x, y, z), r`*`)` — Create a non-hierarchical sphere with name *name* of radius r centered at (x, y, z).

- `gr.cube(`*`name`*`)` — Return an hierarchical box with name *name*. The box should be aligned with the axes of its MCS, with one corner at $(0, 0, 0)$ and the diagonally opposite corner at $(1, 1, 1)$.

- ```
  gr.mesh(name, {
          {v1x, v1y, v1z},
          {v2x, v2y, v2z},
         ...
          {vnx, vny, vnz},
  }, {
          {p11, p12, p13, ...  p1m},
          {p21, p22, p23, ...  p2m},
         ...
          {pn1, pn2, pn3, ...  pnm}
  })
  ```

  Create a polygonal mesh named `name` with the listed vertices and faces. The first list is a list of vertex coordinates, and the second list is a list of polygons. Each vertex is given as an (x, y, z) triple, and each polygon is a list of integer indices into the vertex list. Vertices are indexed starting at 0.

  It may be assumed that polygons are convex and planar. However, polygons may have an arbitrary number of vertices.

- `gr.light({`*`x, y, z`*`}, {`*`r, g, b`*`}, {`*`c0, c1, c2`*`})` — Create a point light source at (x, y, z) of intensity (r, g, b). The attenuation parameters *c0, c1, c2* specify the attenuation for the particular light source according to the formula $1/(c0 + c1 * r + c2 * r^2)$.

- `gr.render(`*`node, filename, w, h, eye, view, up, fov, ambient, lights`*`)` — Raytrace an image of $w \times h$ pixels to *filename*. The camera is to be located at position *eye*, looking in direction *view* with *up* pointing up (all of these quantities are three-vectors). A field-of-view of *fov* degrees is to be used. The ambient light should have an intensity of *ambient* (also a three-vector). All lights to be used in raytracing are listed in *lights*.

The `gr.nh_box` and `gr.nh_sphere` callbacks will allow you to implement a non-hierarchical version of the raytracer, since you can place spheres and boxes in arbitrary locations. Thus you will be able to test aspects of your code such as shading and shadows without necessarily having hierarchical transformations working. Later, you may find it easier to build scenes using `gr.sphere` from Assignment 3 and, say, `gr.cube` for building a unit cube at the origin.
  ```

6.7 Donated Code

The Assignment 4 code is similar to that of Assignment 3. There are a few new files in the source directory:

- `a4.cpp` – Contains the `render()` stub you need to implement.
- `image.cpp` – A class storing rectangular images, supporting PNG saving and loading.
- `light.cpp` – A very simple light class.
- `mesh.cpp` – A simple mesh class.
- `polyroots.cpp` – Robust polynomial root solver. You may optionally use this in your ray-intersection functions.

Furthermore, a number of sample scripts are available in the `data/` directory. Some of these require that you run them in that directory, as they depend on other files found there. A simple Lua reader for the Alias/Wavefront OBJ mesh format is also provided (`readobj.lua`) as well as a sample OBJ file (`cow.obj`).

You may want to merge in some of the changes you have made to Assignment 3. For this assignment, however, you do not need to implement a user interface. Your program will be a command-line program.

In Assignment 3, `scene_lua.cpp` provided a function `import_lua` which returned a scene node after parsing. In this assignment, a function called `run_lua` is provided instead. It will take care of parsing the scene, and call `render()` (from `a4.cpp`) as necessary. Once `run_lua` is done, the program will finish executing.

6.8 Deliverables

Executable: `rt` – This should take a scene file as its argument.

Sample Images: For a complete assignment, generate three image files using the following Lua scripts (which can be found in `/u/gr/cs488/data/A4`):

- `nonhier.lua` – a non-hierarchical scene
- `macho-cows.lua` – a hierarchical scene with instances
- `simple-cows.lua` – a simplified version of `macho-cows.lua`.

Only one of `simple-cows.lua` and `macho-cows.lua` needs to be submitted (although you're welcome to submit both). The image files should be in the PNG format and stored in `nonhier.png`, `macho-cows.png` and `simple-cows.png`. The scripts `macho-cows.lua` and `simple-cows.lua` will fully test all the features of your raytracer except your additional feature. If you do not complete the entire assignment, of course, you will not be able to render all three scenes.

In addition, to demonstrate that you've implemented bounding volumes, you should make a special renderering of either `nonhier.lua` or `macho-cows.lua` where you draw the bounding volumes instead of the polygonal meshes. This image should be stored in `nonhier-bb.png` or `macho-cows-bb.png`.

These image files should be in your `cs488/handin/A4/data` directory.

You will find sample renderings of all three scenes in the image gallery on the course web site.

Don't forget `screenshot01.png`. As usual, it should be in your `cs488/handin/A4` directory. This should be your best image (which should also be in your `cs488/handin/A4/data` directory, but under a different name).

Sample Script: You need to generate a test script called `sample.lua` and render an image from it called `sample.png`. You should place `sample.lua` and all of your images in your `cs488/handin/A4/data` directory. You do not need to submit a special rendering of your sample scene to demonstrate bounding volumes.

This script or another one should demonstrate your "additional feature".

Additional Documentation: Your `README` needs to contain a description of your extra feature and your unique scene(s). If you implement an acceleration feature, provide a switch to turn it on and off (this can be a compile-time switch: provide two executables) and provide comparative timings. If you use external models, please credit where you got them from.

You need to submit at least one image file: `sample.png`. This image should have a resolution of at least 500x500. You may submit additional images if you wish; mention them in your `README`.

If you could not get hierarchical transformations working, submit an image made without them, and mention that you are missing hierarchical transformations in your `README`. You will be severely penalized if we discover that you misrepresented yourself, of course.

There will be a bit of subjective grading of the image created from the data file you create. If the image is extremely good, the instructor may award up to 1 point extra credit. If the image is extremely simple, but tests all features, the TAs may subtract up to one half a mark. If the image does not test all features, more marks may be deducted, since the TAs will not be able to verify that feature.

6.9 Objectives: Assignment 4

Due: Tuesday, November 11th [Week 9].

Name: _____

UserID: _____

Student ID: _____

___ **1:** Objects are visible on the image. This implies that you can generate primary rays, can intersect them with spheres, and can generate a PNG output file.

___ **2:** Cubes and polygonal meshes are properly rendered.

___ **3:** Objects are correctly ordered from back to front.

___ **4:** There is a function that generates a background for the scene without obscuring the view of any of the objects in the scene. This background is on all the generated images.

___ **5:** Diffuse and specular (Phong) lighting has been accomplished.

___ **6:** Shadows have been accomplished.

___ **7:** A script has been supplied that defines and renders a scene different from anyone else's.

___ **8:** Hierarchical transformations operate properly. Spheres and cubes can be transformed with affine transformations.

___ **9:** Bounding volumes (spheres or boxes) have been implemented for polygonal objects as demonstrated by a special renderering as described above.

___ **10:** Some extra improvement has been made to the ray tracer. This may be quite small and range from extra efficiency to extra functionality.[1]

Declaration:

I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

[1] State clearly in your `README` what has been done. If you implement a feature that improves the performance of the ray tracer, you must state in your manual execution times for your ray tracer running both with and without the feature.

Chapter 7

CS488/688 F17 A5: Project

> *"The Dominos pizza number is 745-2222. They deliver and they know where the lab is..."*
>
> Note: This quote is a bit outdated. No food or drink is allowed in the lab.

The proposal for your project is due **Wednesday, November 11th [Week 9]**. It will be marked and returned by the following class, and an updated/revised version is due **Friday, November 20th [Week 10]**. Your project submission is due **Friday, December 4st [Week 12]**, and demos will be held on **TBD [Week 13]** .

7.1 Project Specifications

7.1.1 Purpose

- Demonstrate your grasp of computer graphics.
- Implement a (set of) graphics algorithms of your own choosing.
- Use these graphics algorithms in an interesting context.

7.1.2 Statement

This assignment is the culmination of your efforts. Its primary purpose is to allow you to plan and execute a project of your own creation. So, do something interesting—write a program that does something graphical:

- Develop some graphical model...
- Create a game...
- Animate some scene...
- Explore an interaction...
- Invent a lighting model...

- Make an innovative use of the hardware features...
- Simulate the {physics,dynamics,behaviour} of something...
- or ...

7.1.3 Project Breakdown

The project is broken into three goals. The first two goals are project proposals. We will grade and comment your first proposal and return it to you. **Note that this proposal is worth about half as much to your final grade as a homework assignment.** The second goal is a resubmission of your project proposal, revised according to our comments. We will keep this version of your proposal and use it to grade your project. The third goal is the final project itself.

What we want in your project proposals is an objective list (like those found in the assignments) and supporting documentation that clearly defines these objectives. Half of the rest of your project grade will be based on this objective list. The rest of this chapter describes in detail what we expect for a project and what we want in the proposals.

7.1.4 First Goal (4 pts): Proposal Submission

You are to submit a statement of what you intend to do *in the format of a course assignment.* A hardcopy of this is to be submitted in class on the day noted in important dates.

Please Note: Your proposal will be read seriously by TAs and Instructor, and you will receive criticisms, advice, and suggestions on what you propose. These must be taken into account when submitting a Second Goal (revised) Proposal. **You will not receive any points for the Second Proposal**. The project itself is the means by which the Second Proposal is turned into credit. So it pays you to do a good job the first time around.

Special Note: You must achieve at least one objective to received any credit for your proposal. In particular, if you do not attempt a final project, you will receive 0 marks for your proposal regardless of the grade you achieved on the proposal.

A model in LaTeX format is to be found under `/u/gr/cs488/data/A5` as the file `proposal.tex`. This can be formatted using the commands

```
latex proposal.tex
dvips -f < proposal.dvi > proposal.ps
```

See `man latex` and `man dvips` for further details. There is also a `troff -ms` version in `proposal.tmpl`. Moreover, a sample of a good proposal from a past term, in PostScript form for viewing, is to be found in the file `exampleProposal.ps`

Your proposal should including the following sections:

- A **Topics/Purpose** list.
- A general, introductory **Statement** of the nature of the project. For a ray tracer, you should describe your scene, saying what features will be needed to achieve specific effects.

- A **Technical Outline** section surveying the important data structures and algorithms that will be necessary to achieve the goals, and (for ray tracing projects) lists the new commands that will need to be added to the input language.
- A **Bibliography** of a small number of papers and/or books that will be consulted, with a comment about the relevance of each to the project.
- An **Objectives** sheet containing ten different points upon which the achievement of the stated goals are to be judged. Note that if you are proposing a ray tracing project, then you should list the extra objective you implemented for A4 at the bottom of this list.

BE CAREFUL! While part of your grade will be based upon your success at reaching your goals, another part of your grade will be based upon your intelligence, understanding, comprehension, and good sense at setting goals that are neither too hard to be achieved nor too easy to be significant. Inventiveness and originality will count as well.

Grading

The TAs and instructor give points to your first proposal subjectively, by comparing the proposals against each other and against expectations. What we want to see in your proposal is that you have investigated what you plan to do for your project, and that you have decided upon a reasonable project. To receive full marks for the first proposal, you need to have

1. A clear description of your project;
2. An outline of the technical details that indicate that you understand the issues involved;
3. Appropriate technical references from reference books, texts, journals, and conference literature;
4. A list of non-trivial, pertinent, obtainable objectives.
5. Some amount of individuality in what you propose

If you are vague about your plans or objectives, if you are too ambitious or have unrealistic objectives, or if it is clear that you have not read up on your chosen subject, you will lose points.

Special note for ray tracing proposals: In your proposal, be sure to mention what your extra objective was for Assignment 4.

7.1.5 Second Goal: Revised Proposal

A revised copy and grading sheet is due after you received TA/Instructor feedback. Again, only a hardcopy is to be submitted. The objective list of this document will be used to determine part of the grade for the third goal of your project, and thus it will be retained by the instructor, and as such will contribute to your project grade. However, no points will be explicitly assigned to this goal.

By the time this is due, you should have received your graded assignment 4. If you intended to do a ray tracer project and did not get 9/10 on the assignment, you should switch to an OpenGL project. If you have to switch projects, make sure you discuss your new objective list with your instructor.

7.1.6 Third Goal (20 pts): Project Completion

You will submit physical documentation for your project. On the due date, your documentation is to be turned in to the instructor or TA. You can submit your documentation to the Computer Science office during normal working hours (8:30-12:00,1:00-4:30). **This documentation must be submitted in a bound format.** Five points will be deducted if it is not bound. Most types of binding are acceptable (three-ring binder, Cerlox, clip-on, etc.); basically, it should have a cover and be connected together (but *not* stapled or paper clipped). Projects will be returned and "medals" awarded at the final exam.

The documentation you hand in should comprise:

1. The standard material listed in the **Documentation Submission** portion of this handout, with particular attention being given to the **Manual** section, which should contain a comprehensive guide to the running of the project's program and to the formats, where appropriate, of its input, command line, interaction, output, and procedures for creating/viewing images.

   You are also required to have the usual `README` telling us how to run your program. In addition, at the end of your `README`, **you should list your objectives**.

2. An extra **Implementation** section that describes the software design considerations, including brief descriptions, where appropriate, about

   - Algorithms, data structures, and complexities,
   - Modularity, data abstraction and encapsulation,
   - Platform and system dependence or independence, global constants and configurability,
   - Input/output syntax, Lua extensions if any, pre- and post-processing,
   - Data and code sources, network and literature resources, system and local utilities, the re-use and adaptation of existing code,
   - Intercommunication, shell scripts, pipes, sockets, intermediate files, parallelism and task delegation,
   - Coding style, debugging approach and utilities, version management, testing and verifying practices,
   - Caveats, bugs, cautions, unexplored areas, assumptions, future possibilities.

3. For graduate students only *a short* report on the main points of the most important bibliographic reference(s) used for the project.

Note that the documentation you submit strongly influences your subjective mark on the project. In the past, the projects receiving the highest subjective marks had excellent documentation. Often, two projects would have approximately the same technical merit, but one would receive a much better mark due to differences in documentation. Here's a brief rundown of what we look for in documentation:

- A discussion of the interesting technical points of the project. You should note what is interesting and what is required to implement the idea.

- A map of the code. We don't want manual pages, but would like pointers to where we can find the various features you have implemented.
- References. You should note the key technical references for your project.
- Acknowledgments. You should credit any code written by other people and acknowledge people who have helped you.

To give you an example of a reasonable level of documentation, look at the sample report PostScript file to be found under `/u/gr/cs488/data/A5`. Graduate documentation is to be done in the same style, but it has to include more extensive details on implementation and a summary about any related prior work in the published literature.

Previously Written Code, Use of Other People's Code

Your project is to be code you have written expressly for the project. As such, you should not in general use code you wrote in previous terms, nor should you use other people's code in your project. However, there may be cases where you want to build on pre-existing code. If you find yourself in such a situation (whether it is code you wrote in previous terms or if it is someone else's code), you should discuss the matter with the instructor. Further, you should document in your project write-up any previously written code or code you get elsewhere. Failure to document use of such code will be considered *cheating*, even if you wrote the code yourself.

In particular, if you wrote something in previous terms that seems appropriate for your CS 488/688 project, you *may not* submit it as your CS 488/688 project. You may, however, use it as a starting point for your project; you should discuss such a matter with the instructor for the course before submitting your project proposal.

Grading

The remainder of the grade for the third goal will be based upon a subjective assessment by us, the instructor and TA(s), of how your project ranked against the others submitted this term, as well as projects like yours submitted on past terms. In total, the grade will be assigned somewhat like the judging in the Olympics for figure skating. Your list of objectives provides the scores for the "required elements," and our assessment provides the scores for the "individual merit." The individual merit judgment will be arrived at by considering the the four components of "artistic and/or innovative content," "technical depth," "software design," and "quality of documentation." This judgment will, necessarily, need to be subjective, given the wide diversity of projects. In the past we have used such criteria as:

- Artistic. Visual design and aesthetics. We will also use this category to reward humour, cleverness, originality, inventiveness, polish.
- Technical. Algorithms, mathematics, physical or optical simulation, data structures.
- Hardware techniques. Use of hardware features above and beyond those in the assignments.

- User interface. Quality of interactivity and feedback, user friendliness. Individual design of user-interface and interactive tools. Note that a user interface that is essentially ones from the assignments will net you no marks in this category.
- Code. Comments, modularity, code design.
- Documentation. Quality, thoroughness, and depth of documentation, background references, literature summaries, breadth of resources employed.

  Special Note: You will get 0 subjective marks for documentation if:

  1. your files are not where they are supposed to be, or
  2. your writeup (README + Documentation) is insufficient for us to determine how to run/use your project.
- Difficulty. This category is used to reward exceptionally difficult projects.

These criteria are an example, and they may change to suit the content mix of each term's projects. Nobody's project is expected to hit all these criteria, so we are prepared to give up to 3 points in any category. We will stop when we get to 10 subjective points, although achieving 10 points is **extremely rare**. On the average, good projects receive 4–6 subjective points, and only unusually, outstandingly, remarkably excellent projects are in the 8–10 range. The TA(s) and instructor will be as fair as possible, but standards will be high, and a "perfect 10.0" will not be given lightly—we'll probably not award it to more than one person, if that many. We are looking for polish, depth, professionalism. We are trying to find the remarkable, and locate evidence of care, thought, effort, and skill that puts the project above one that just managed to get its technical objectives.

Note that we may also give out down to -1 points in any category. This is for things you have learned during the term that you should have applied to your project, but you didn't. For example, if you do not have shading, you may get -1 in a category such as graphics techniques.

The instructor will tell you how your subjective mark was determined if you ask. However, the subjective mark is subjective; it's our opinion. If we overlooked something in our marking, then we may increase your subjective mark. However, if it is a matter of your opinion vs. our opinion, then we won't change your subjective mark. I.e., if you feel your user interface is easy to use, but we don't, then we won't change your mark. If your friends thought your project was the coolest thing they've ever seen, great! But we won't change your subjective mark.

The highest three total achievements (objective plus subjective) will be distinguished by the special awarding of a "gold," a "silver," and a "bronze" prize to be given out at the final exam. Honorable mentions may be awarded to other projects that we feel deserve special recognition.

A note about "pure" rendering (i.e., ray tracing) projects: if you look at the above list, you will see that you will easily receive points for algorithms, etc, but can not receive any for user interface issues. You should therefore pay particular attention to the following categories: Visual Design and Aesthetics (make a *very* nice image) and Documentation. In addition to written documentation, you are expected to have test images that exhibit the features you have implemented. It is insufficient to just submit your "nice image". Further, if you implement an optimisation technique, you should give timing comparisons for the renderer running both with and without your optimisations. Your documentation should state where in your code these efficiency improvements are implemented. If any background references were used, summarize these in your documentation.

7.1.7 Demonstrations

Students will demonstrate their projects on **TBD [Week 13]** ; either a sign-up sheet will be posted on the instructor's door or an online sign-up system will be used. The purpose of such a demo is for you to show that you have met your objectives, to show any extra features you have added, and to make clear what particular strong points you feel your project offers in any subjective categories. Therefore, you should base your demonstration around your objectives and around any of its possible subjective merits. Your objective points will be awarded during the demo. The subjective points are determined by the Instructor and the TA(s) as they explore your project again at a later time.

Only the TAs, the instructor, and the student giving the demo will be allowed in the lab during the demonstration. Approximately fifteen minutes will be given for each demo. You should rehearse your demonstration ahead of time, and make sure all your data files and executables are set up properly beforehand.

Be sure that your demo illustrates that your objectives have been met (you do not have to demonstrate non-graphical objectives such as optimization code).

Project Demo Requirements

- Make sure you've tested your program. Believe it or not, one person in Fall 2000 had never run his program before the demonstration. He did not get any of his objective marks.
- There should be no reason to recompile or edit things during the demo.
- For interactive projects:
  - You should be able to toggle features such as texture mapping, shadows, reflection etc in your program. Otherwise we may give you a 0 for your objective.
  - You may want to implement a cheat/god mode of your game to demonstrate some objectives. For example, if you implement particle systems that shows up when you hit 5 moving targets in a row, but it is very hard to aim, a cheat mode where you never miss will be very useful and maybe even necessary.
  - If your project was implemented at home, and some objectives did not work properly on the school machines, take screen shots of those feature and show them during the demo so you can get half marks for those objectives.
- For raytracing projects:
  - Each objective should have 1 or 2 images (for comparison) to illustrate you have completed that objective. These images should not be your final scene and should preferably demonstrate 1 objective only. Good raytracers generally have 10-20 images in addition to their final images. Rehearse showing your images. You don't want to search to find the image that shows your objective during the demo.
  - Show your best scene last.

7.2 Collected Wisdom

The following are our thoughts on various projects: what worked, what didn't work, etc., based on previous projects.

7.2.1 Finding a Project

There are several ways to find a project. First, you may have seen something in the course that you want to learn more about. This is the ideal case: just follow up on what you are interested in and see if you can make it into a project. The second way to find a project is to look for an interesting topic in a textbook, or through past issues of the conference proceedings for SIGGRAPH (older issues are published as special issues of the *Computer Graphics* journal) or *Graphics Interface*—both of which are in the library.

In both cases, you should try to find a project that is neither too hard nor too easy. The first step in deciding if a project is just right is to make up a list of objectives (Goal 1 of the project). Further, you should look at the list of subjective marks upon which we grade your project, and try to decide what subjective marks you could hope to get from your project.

As a general comment, think of the final overall effect you want from your project and come up with objectives for that effect, rather than think up objectives and build a project around it. For example, rather than say "I want to walk through a L-system forest on a fractal terrain," it'd be better to say "I want to write a golf game with trees and hills." The objectives are (roughly) the same, but the latter results in a cohesive project, while the former results in an unsatisfying project even if all the objectives are met.

If you have questions about what makes a reasonable project, ask the instructor or the TA(s). In the past, students who discussed their project with the instructor **before** submitting their first proposal would generate much better proposals and ultimately have better projects than those who did not see the instructor.

Once you find a project topic, you can use the ACM SIGGRAPH online bibliography to find papers on the topic. This bibliography on is available the World Wide Web at

http://www.siggraph.org/publications/bibliography/bibliography.html

7.2.2 Projects to be Wary of

In the past, some projects have worked better than other projects. Here's some projects that we won't accept unless you extend them in a novel way:

- Rubik's cube. This is just hierarchical modeling and transformations. Solving the puzzle is not a graphics problem.
- Fractals. While it is fine to use fractals to model something in your project, generally there isn't enough here to make a full project.
- L-systems. Same problem as fractals.
- Particle Systems. Same problem as fractals.

- Fireworks. Fireworks are a simple form of particle system and is not enough to make a project by itself.
- Solar Systems. We've had many of these in the past, and all were boring. This is a hard project to make interesting.

And here are some examples of projects that often (but not always) fare poorly when it comes to grading:

- Human/animal animation. It's hard to make creatures walk, dance, juggle, do combat, etc.
- Space games. Usually, too much emphasis is placed on the game aspects. The problem in developing good objectives is that objectives that relate to graphics are usually harder than they look. Further, these projects often get low subjective marks.
- Maze games. It's not that this is a bad project, but we've seen a lot of them so our standards are high.
- Flocking. This is harder than it looks. Further, it is hard to demo (the flock tends to fly off the screen), so if you do flocking, be sure to have special camera modes that track the flock.
- Modellers. A general purpose modeller is hard to implement. If you want to implement a modeller, design it to build something specific and expect to spend a lot of time using your modeller to man an interesting model.
- Keyframe animators. Same problem as modellers.

7.2.3 OpenGL or Ray Tracing?

One of the big decisions you need to make when deciding upon a project is whether to do an OpenGL interactive project, or a ray tracing project. It is simpler to think up a ray tracing project: You just have to make a list of extensions to the ray tracer. Interactive projects are a bit harder to devise: You need to think of an idea, determine what's interesting, and work it into a project.

Both are acceptable as projects. And while students have won awards (see below) for both types of projects, past experience indicates that interactive projects (on average) receive higher grades. Although this may in part be a reflection upon the type of students (in general) who choose ray tracing projects, there is one definite difficulty with ray tracing projects: You have to write an excellent ray tracer to get a good subjective score. And here is a hint of what we mean by excellent: **You cannot do a ray tracer project unless you got at least 9/10 for assignment 4.**

For a ray tracing project, you should think of an image/scene you want to render. Then develop your project around this scene: What features does the ray tracer need to render this scene? What features will you need to make it look realistic? Then, as you extend your A4 ray tracer, you can develop the model for your scene as you go. You should mention your scene in your proposal, and while it will be ideal if you show us a picture of the scene at your demo, it is not a requirement that you do so.

Most ray tracers will need CSG and some form of texture mapping to make a nice picture – and you can't get good subjective mark without one. However, to be a project with substance, you would need to make fairly complete/general implementations of CSG and texture mapping. And texture mapping (mapping a 2D pattern or image onto a surface) is difficult to do well, without objectionable distortion. Hence, it may be quite difficult to get good subjective marks without close attention to considerable mathematical and implementation detail. Be sure you know what the issues are before you promise to do something, particularly in rendering projects.

As discussed later, a nice scene is critical for the subjective marks of a raytracing project. We don't want to just see shapes, we want to see an actual scene. If you don't have a fine arts background, we recommend against doing scenes with just a few simple objects, i.e., you should model real life objects in a real life scene.

7.2.4 About the proposal...

The goals of your proposal are (a) to tell us what your project is and (b) convince that your project is reasonable in the sense that it's not to hard and not too easy.

Technical Outline

In this section, you need to explain the important data structures and algorithms that will be necessary to achieve your objectives. For raytracing projects, include the commands that must be added to the input language.

How well do you need to explain the data structures and algorithms? Well, you must convince us that you understand what is involved with each objective. For example, if you have bump mapping as one of your objectives, you must explain to us how you intend to map the bump map to each primitive, how the normals are perturbed, etc. If you are implementing something from a paper, it is not enough to refer to the paper, we expect you to list out the steps in the algorithm and include necessary equations. For raytracing projects, explain how your objectives will be used in your scene.

When we read your objective list, we will refer to your technical outline for details. A good technical outline will tell us exactly how you intend to achieve each objective.

Objectives

What are good, reasonable objectives? Briefly, an objective is a unit that contributes one fundamental, essential goal to your project. On the average, an objective is roughly 1/10 of your work. In a 2-3 week development time, it represents 1-2 days of planning, implementation, checking, and correction. A poorly done proposal is often characterized by attempting way too much, and that derives from not understanding clearly what is involved – either in terms of objectives or in the difficulty of each objective. You should also think about what kind of subjective marks you can get. If a project is too difficult, you may have trouble getting your objective marks.

Examples of poor objectives for extending the ray tracer follows:

1. Add spline surfaces
2. Add anti-aliasing

The first objective is too difficult and should be broken down. Both are too vague, although if adequately described in the Technical Outline then they might be okay. In this particular project, the person who wrote those objectives (no, we did not make them up) is revealing that he/she hasn't a clue about what any of those words means.

Here are good things to avoid in your objectives list:

- Code is well organised.
- Comments are good.
- Program executes correctly.

These are expected to be true as a minimum, not as objectives worth special mention. You should also avoid such objectives as

- Objects do not fly apart or distort under transformations.
- Picking works correctly.
- Program supports multiple views.

We went through these in the earlier assignments. Of course you can do this. It's not worth including as an objective. Finally, here are some objectives that are too vague:

- Interesting interaction.
- Useful feedback given on screen.
- Good use of colour.

Can't you be a bit more specific? What is the meaning of "interesting?" Who decides what is "useful" or "good?" How does the TA assess what goal has been reached?

In short: an objective should be precisely stated, clearly understood, capable of unambiguous determination about whether and to what degree it has been met. If you are using words like 'nice' or 'easy' or 'useful' or 'simple' or 'interesting' or 'realistic' in your "objective", then it's probably subjective and should be changed. Your objectives should not rehash fundamental objectives of past assignments, and they should not address basic software engineering issues that every good CS student should have mastered by now. Concentrate on objectives that are addressed specifically to the unique points of your individual project. An objective is given as a short, simple, declarative statement. For example, we can rewrite the two raytracing objectives given earlier as such:

1. Bézier surfaces is subdivided into a polygon mesh.
2. Phong shading is added.
3. Super sampling is used for anti-aliasing

One other thing to be careful of when making goals: your goals should have some technical graphics content. We commonly allow one of each of the following as objectives

- A modelling objective — this can be your final raytraced image or your OpenGL characters and scene.
- An animation objective — if you are implementing a keyframe system, creating animations with it can be one objective.
- A user interface objective — this objective can be used for some slightly complex interaction such as controlling a plane or editing a tree structure. Manipulating the camera, picking, etc, cannot count in your U.I. objective since they are already covered in your assignments. *Note: Be wary if your project needs a lot of menus and dialogs. These are very easy to implement, and so are not worth a lot of subjective points, but they can take up a lot of time better spent on achieving other objectives and fine-tuning your project.*
- An artificial intelligence objective — this objective does not really have technical graphics content, but may be useful for games. If you want an A.I. objective, it better be a good one!

Unacceptable objectives are "feature" goals without graphics content. This commonly happens with game projects. For such projects, displaying a numeric score, or giving the player extra life when a food pod is consumed may all be nice game features, but none of them are acceptable as objectives for a graphics project.

Bibliography

An excellent way to prepare for a good proposal, and to continue from there to a successful project, is to give evidence that you have informed yourself about the issues and the techniques. You do this through reading some reference material. An ideal preliminary source of ideas can be provided by looking in the index or table of contents of the texts, both required and recommended, for topics of interest. These texts will often point off to the literature for further reading. Including **specific** pertinent references to the literature in your proposals is a good way to show that you know what you are doing. For example, on 3D textures,

Bad Bibliography:

Computer Graphics, C Version, Second Edition, Hearn and Baker, Prentice Hall, 1997

(This just lists a book containing information on hundreds of topics.)

Good Bibliography:

1. **Computer Graphics, Principles and Practice, Second Edition**, Foley, vanDam, Feiner, and Hughes, Addison-Wesley, 1990, pp. 1015–1018.
2. *An Image Synthesizer*, Perlin, SIGGRAPH '85 Proceedings, pp. 287–296.
3. *Solid Texturing of Complex Surfaces*, Peachy, SIGGRAPH '85 Proceedings, pp. 279–286.

(This lists specific pages on the subject in a reference text, and it additionally lists two landmark conference articles on the subject.)

If you use reference material on the internet, you should certainly list it in your bibliography, *but a website alone is not an acceptable reference.* Websites detailing algorithms often list the source, so make sure you check those out as well.

"Life is so much simpler and easier now that graphics is over!"

– A former CS 688 student.

"I dunno – I had withdrawal symptoms."

– A former CS 488 student, after reading the previous quote.