CS 371 Project 0:

Cubes

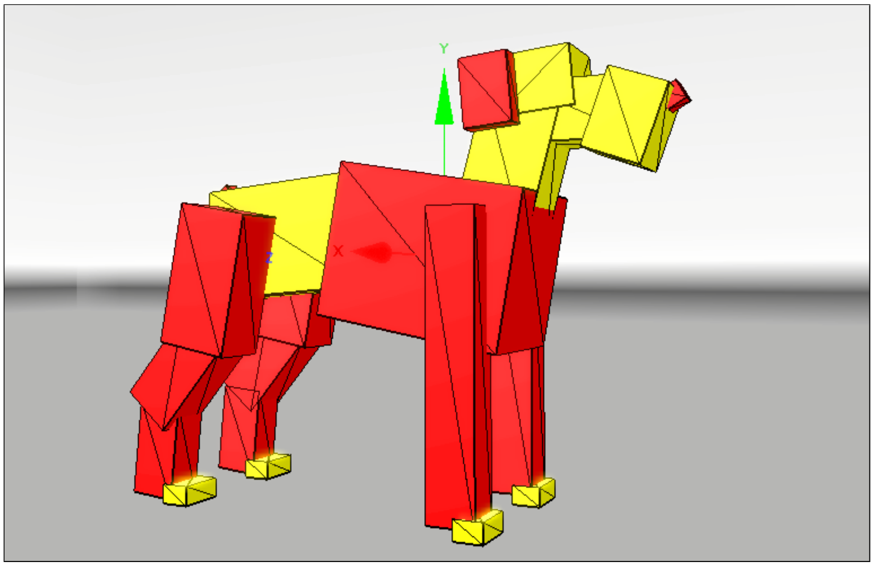

Figure 1: *A dog modeled with translated, rotated, and scaled cubes. By the end of this project you'll know how to create scenes like this programmatically and write an interactive real-time 3D renderer for viewing them.*

Contents

1 Introduction

1.1 Overview

Welcome to your first CS371 Project! In this project you'll write a C++ program that displays a set of 3D cubes and then write a short report. The code that you write *this* week will be the starting point for the new project *next* week, so take care to structure the program in a flexible manner and be sure to document your source clearly.

For the other projects this semester, you will read the specification and start work before scheduled lab. This project is unique. We're going to go through the handout and begin implementation together during the first lab session. This project also introduces a number of tools and libraries that may be new to you. For those reason, the handout is really long. It explicitly walks you through most of the steps in the project. As you progress through the course you will learn to work directly from a technical specification, primary research sources, and reference documents, so you'll need less direction and detail in the handouts.

Note that I hyperlinked the section numbers, figure numbers, citations, and URLs in this document to help you navigate quickly in the PDF version. You should return the favor by structure your project documentation with links like this, most of which will be done for you by Doxygen if you follow the formatting guidelines.

1.2 Educational Goals

In this project, you'll gain familiarity with:

1. **Some 3D modeling conventions:**
   (a) Coordinate system and units
   (b) Positioning objects in 3D space
   (c) A first-person camera controller
   (d) The Model/Entity design pattern
2. **Some CS371 software development tools:**
   (a) The C++ programming language
   (b) The Subversion (svn) revision control system
   (c) The G3D library
   (d) The Doxygen documentation generation program
3. **Programming in the large:**
   (a) Automatic memory management
   (b) Overview documentation
   (c) Entry point documentation

1.3 Schedule

Out:	Tuesday,	September 7	
Due:	Monday,	October 13,	10:00 pm

This is a moderate, solo project. The "moderate" rating is because you're learning to use a new programming and documentation environment at the same time that you're implementing a specification.

This warmup project is structured slightly differently than the other projects this semester. For most projects, you'll start working on Tuesday or Wednesday. In lab on Thursday we'll complete some flexibly-structured exercises designed to accelerate your progress and get you to (and maybe through) the crux of each project. You will then have until the following Monday night to complete the project.

For this project, don't start before the scheduled lab session. We'll begin it as a class in lab on Wednesday, September 8th. You will then complete the project at your own convenience. I encourage you to ask questions outside of the scheduled lab times by e-mail, during office hours, or in lecture.

As a reference, my solution for this project was about 250 statements and 150 comment lines as reported by iCompile, including the Doxygen comments that generate the report. More than half of the code was in `Scene.cpp` for creating the three required scenes. If at some point your implementation looks like it will be significantly longer or shorter than that, come talk to me because you may have gone a bit down the wrong track.

Track how much time you spend on this project outside class. You're required to include this in your final report.

If you haven't completed the report and everything except the custom scene within three hours of work after the scheduled lab, stop working and talk to me immediately. In that case you are putting your effort into the wrong part, or I didn't explain something clearly enough. The entire project should take at most six hours outside of lab to complete.

1.4 Honor Code & Rules

You are encouraged to talk to other students and share strategies and programming techniques but should not look at each other's code directly. The honor code policy for CS371 is designed to encourage more collaboration than in other courses. In fact, collaboration with other students is an important factor in your class participation grade. Collaboration means sharing appropriate information, code, and data with others in the class, including people who *aren't* your assigned partner. See the *Welcome to Computer Graphics* document from the first lecture for the explicit course policies.

For this project only, you are **not permitted** to look at the sample projects in the G3D distribution. You may not look at or invoke the `G3D::GEntity` class or the `G3D::ArticulatedModel::createCornellBox` method. You may look at and use the rest of the G3D source code.

2 Specification

For each project, you will submit **source code**, **documentation**, and a **report** that includes figures and data. These are unified within the source code and submitted through the revision control system–I will grade whatever is checked in at the time of the deadline. Note that I'm evaluating these three documents, not just the functionality of your program.

Create your class, method, and function documentation as specially-formatted Doxygen comments immediately before the element being described inside the C++ header (`.h`) files. Prepare your report as a large Doxygen comment in a file ending with `.dox`. From your report, include links to relevant code elements and to images and videos that you have prepared.

1. Build a program to load and visualize small scenes with an interactive camera.

2. Create the following scenes using only `cube.ifs` and the `whiteroom` environment map files:

   (a) A single, white 1 m^3 cube centered 1 m along the positive x-axis and rotated 45 degrees about the vertical axis.

   (b) A model of the Cornell Box pictured in Figure 4.

   (c) A visually interesting scene of your own design.

3. Use high-level library routines to abstract the operating system, file format, and immediate-mode rendering, as detailed in this document.

4. Create overview and entry point documentation for your software using Doxygen.

5. Create the report described in Section 2.1.

2.1 Report

1. Make simple, isometric view, labelled axis-diagrams of the 2D coordinate and 3D coordinate systems (by hand; don't write code for this), and include it in your report. On the 3D coordinate system, show the direction of increase of the yaw, roll, and pitch angles. I would personally use PowerPoint to create the diagram and then convert it to a PNG by pressing command-3 on the Mac and selecting the relevant area on the screen. However, you may use any reasonable method that you like, including SVG and ASCII art, so long as your solution renders correctly under Safari.

Tip: The Tools document contains sections on the 2D and 3D coordinate systems.

2. Assume that someone who doesn't know anything about G3D or your program is going to have to modify it in the future. Describe the structure of your program for this person in your report, with links to major classes and methods. This should only take about one paragraph of space.

3. Include images of the single scene, the required multi-cube scene, and your custom scene. Crop these appropriately using Photoshop, and link a thumbnail to the actual image. Put the actual image files in the `doc-files` directory and link from smaller versions of them. Look at the HTML source for the "Cube Maps" section of the "Index of Data Files" page of the G3D manual using your web browser to see how to do this.

4. **Questions.** Knowing how to use documentation, experimentation, and reverse engineering to discover how a system works are important skills. In this lab you copied a lot of code that I wrote. To gain mastery over that code, figure out the answers to the following questions and write them in your report. You're going to have to get your hands dirty on this–the answers aren't just sitting there. Don't share the answers with your classmates, but I encourage you to discuss strategies for finding them.

   (a) What are the differences between the `Scene*` and `Scene::Ref` types?
   (b) What is the `ICE_EXTRA_SOURCE` environment variable for?
   (c) What is the `INCLUDE` environment variable for?
   (d) Why did I tell you to put your initialization code into `App::onInit` instead of constructor `App::App`? (There are many reasons. Try throwing an exception from each, and consider the implications of throwing an exception from a class's constructor.)
   (e) What invokes `App::onInit`, `App::onPose`, and `App::onGraphics`?
   (f) Where is the file "cube.ifs" stored on the file system? What made the `System::findDataFile` look there?

Tip: When trying to understand a library or language feature, imagine yourself in the API or compiler writer's place. How would *you* have implemented it? What constraints force that design?

5. **Feedback.** Your feedback is important to me for tuning the upcoming projects and lectures. In general, please let me know how the course is going for you and how I can make this the best experience for you. On this and every future project, report the following specifically (you get points for answering these questions!):

   (a) How many hours you spent **outside** of class on this project on **required** elements, i.e., the minimum needed to satisfy the specification.
   (b) How many additional hours you spent outside of class on this project on **optional** elements, such as polishing your custom scene or extreme formatting of the report.
   (c) Rate the difficulty of this project for this point in a 300-level course as: too easy, easy, moderate, challenging, or too hard.
   (d) What did you learn on this project (very briefly)? Rate the educational value relative to the time invested from 1 (low) to 5 (high).

3 Evaluation Process and Metrics

To evaluate your project, I will check your project out from Subversion as of the deadline time. I will then run `icompile --doc` to generate the final report and documentation. I will read sections of your source code, the report in the `index.html` page generated by Doxygen, and sections of your documentation as generated by Doxygen. I may run your program, but I will primarily investigate its functionality by the description that you provide in the report. Note under this scheme, that the artifacts from your creation of and experimentation with the program are more important than the executable program itself. For many projects you can receive a favorable evaluation even if your program does not compile or execute.

As described in the *Welcome to Computer Graphics* document, I will evaluate your project in several categories:

- Mathematical (algorithm, geometry, physics) correctness
- Adherence to the specification
- Program quality
- Report quality

Some questions I consider when evaluating the source code are: Is it possible for someone unfamiliar with it to find specific routines quickly? Is the code easy to understand? Does it make good tradeoffs between efficiency, clarity, and flexibility? Are data structures used effectively? Are the algorithms correct? Are the geometry and physics correct?

When evaluating the report, I consider: Do the experiments adequately explore the correctness, performance, robustness, and parameter space of the algorithm? Are known bugs made clear, along with how you tried to solve them? Are appropriate sources cited for algorithms and code? Does the overview documentation guide a reader to the relevant source code documentation? Is the architecture of the program clear?

The report and code should both be as concise as possible without compromising clarity. Use the papers we've read as examples of how to describe experiments compactly.

Most students want to create a really impressive 3D scene for the "visually impressive" screenshot mentioned in the specification. Keep in mind that I value your process and presentation more than your program's functionality. To get an "A" you need to answer all of the questions from the specifications, format your report cleanly, provide appropriate entry point documentation, demonstrate effective use of the Model/Entity design pattern, and do the *minimum necessary* to satisfy the specification. Going above and beyond the specification is personally satisfying, but earns you no additional points and will cost you points if you do so at the expense of required elements!

I require you to report the number of hours that you spent on the project. That number will not affect your grade. If you're spending a lot more time than others I will suggest some ways to improve your workflow. Everyone is spending more time than I expected I will reduce the requirements for future projects. If you're spending much less time than I expected I'll suggest some other directions you might optionally explore if you want to learn more about graphics.

4 Walkthrough

This lab contains detailed instructions for setting up your program because it is your first time using the development environment and libraries. Future projects will include a specification and some advice, but you create the software design and implementation plan yourself.

Where this walkthrough says to enter specific code, please actually type it–do not copy from the PDF and paste it into your editor. Typing the code yourself should prompt you think about what it means, and if you make a mistake will give you an opportunity to debug it.

Figure 2: *The OS X terminal window icon.*

4.1 Command Line C++ Programming on OS X

1. **Open an OS X terminal window.** The corresponding dock icon is shown in Figure 2.

2. **Update your `.bashrc` file**. Run:

   `/usr/mac-cs-local/bin/check_login`

   It may tell you to then run additional commands.

3. **Configure your compilation environment**. Open `~/.local_bashrc` in your favorite editor (mine is Emacs) and ensure that your environment variables contain the CS371 paths. These should look something like:

```
G3D9=/usr/mac-cs-local/share/cs371/G3D
export INCLUDE=$G3D9/include:$INCLUDE
export LIBRARY=$G3D9/lib:$LIBRARY
export PATH=/usr/mac-cs-local/share/cs371:$G3D9/bin:
    /usr/texbin:/opt/local/bin:$PATH
export G3D9DATA=$G3D9/data
export ICE_EXTRA_SOURCE=$G3D9/source/GLG3D.lib/source:
    $G3D9/source/GLG3D.lib/include/GLG3D:
    $G3D9/source/G3D.lib/source:
    $G3D9/source/G3D.lib/include/G3D:$ICE_EXTRA_SOURCE
```

   Note that there are no spaces around the equal signs and that paths are separated by colons. The `PATH` and `ICE_EXTRA_SOURCE` variables should each be entirely on one line–I reformatted those to fit on this page.

Tip: Take a few minutes to set your prompt, screen brightness, key repeat rate, `.emacs` file, Safari bookmarks, and Dock configuration. Time spent making your development environment efficient is well spent!

4. **Configure your subversion environment**. At the command line, execute:

```
svn status
```

and ignore the warning that it prints.

This will create a ~/.subversion directory. Open ~/.subversion/config. Search for the global-ignores line and replace it with:

```
global-ignores = *.o *.lo *.la *.al .libs *.so
   *.so.[0-9]* *.a *.pyc *.pyo *.rej *~ #*#
   .#* .*.swp .DS_Store g3d-license.txt log.txt
    temp tmp .ice-tmp build
```

This should all be on one line; I had to break the line here because it was too long to print. This setting tells the revision control system to ignore certain generated files and directories. Everything up to g3d-license.txt is probably already in the file but commented out.

5. **Go to the scratch directory**. In this course, we keep our code under revision control on a server. During a programming session, we always check out that code to the local disk, and then check it back into the server at the end of the session. You want your code on the server between sessions because it enables collaboration on the pair-programming assignments, keeps your data safe in the event that something happens to the computer you're working on, and allows you to revert to a previous version if you make a mistake. You want to compile on the local scratch disk instead of your home directory because your home directory is on the network and is very slow. To get to the scratch disk on the Mac, type:

```
cd /local-scratch
```

6. **Check out your project directory from Subversion.**. For each project I will set up a Subversion directory for you. For the first project the name is simply cubes-$USER, where you can type your username in place of $USER or just allow the OS X shell to replace the environment variable for you.

   You should have already received your Subversion account name and password by e-mail. Your username is the same as your Unix and Mac OS account name. Your password is not the same, and you cannot change it yourself– tell me right away if your password has been compromised and I will give you a new one.

   The commands to check out the first project are:

```
svn co svn://graphics-svn.cs.williams.edu/371/0-Cubes/cubes-$USER

cd cubes-$USER
```

   Since there's nothing in your project yet, this will just make a directory with a .svn subdirectory. Do not ever copy, delete, or directly manipulate the .svn subdirectory.

7. **Write a small program in Emacs.** You used the C programming language previously in CS237 and possibly other courses. We'll go through a quick refresher and introduce the debugger. Start by opening Emacs and entering the following program. When you're done, save it as `main.cpp`, but do not quit Emacs.

```
#include <stdio.h>

void f() {
     throw "Exception";
}

int main(const int argc, const char* argv[]) {
    // f();
    printf("Hello, world!\n");
    return 0;
}
```

Tip: "emacs –nw" runs Emacs in a terminal window, launches fast, and runs over SSH. "emacs" launches Xemacs, which lacks those nice properties but gives you menu bars.

Tip: This might be a good time to look up the Emacs commands for splitting and unifying panes, and switching buffers if you've forgotten them.

8. **Compile with g++.**: Open a second view pane inside Emacs using "C-X 2". Do *not* open a second terminal window. Create a shell under Emacs using "M-x shell". From that shell, compile your program using the command:

```
g++ -g main.cpp -o hello-world
```

Run your program by executing `hello-world` at the command line. It should print..."Hello, world!".

9. **Run under gdb.** We're going to see how to run a program under the command-line debugger and perform basic operations. Debuggers are most useful when your program is doing something wrong, so we have to break the program. Uncomment the line in the body of `main()` that calls function `f()` and recompile your program. Now, launch the debugger with

```
gdb hello-world
```

(a) Press "r" to run your program.

(b) When it crashes, type "bt" to see a backtrace. It should look like:

```
(gdb) bt
#0  0x00007fff86db83d6 in __kill ()
#1  0x00007fff86e58972 in abort ()
#2  0x00007fff885455d2 in __gnu_cxx::
    __verbose_terminate_handler ()
#3  0x00007fff88543ae1 in __cxxabiv1::__terminate ()
#4  0x00007fff88543b16 in std::terminate ()
#5  0x00007fff88543bfc in __cxa_throw ()
#6  0x0000000100000e8e in f () at main.cpp:4
#7  0x0000000100000ea2 in main (argc=1, argv=0x7fff5fbff4b8)
    at main.cpp:8
```

(c) Type "frame 6" to select the `f` stack frame.

(d) Type "list" to see the source code around the active line (you can also look at line 4 of `main.cpp`, since the debugger told you that is where the problem was.) It will show you the code that triggered the exception.

(e) Now switch stack frame #7 so we can look at some variables.

(f) Type "print argc" to look at `argc`. Since `argc` was a formal parameters for the function, it is also printed in the back trace directly.

(g) Quit the debugger by typing "q".

10. **Compile with iCompile.** Fix your program by commenting out the call to `f()` again. You could continue to directly invoke g++ for the rest of your time in 371, however the g++ command line gets complicated very quickly when we write more sophisticated programs. For example, the command line to compile the project you'll complete this week might look like:

```
g++ -D_DEBUG -g -D__cdecl= -D__stdcall= -D__fastcall=
  -fasm-blocks -arch i686 -msse3 -mfpmath=sse -pipe
  -Wall -Wformat=2 -Wno-format-nonliteral
  -Wno-deprecated-declarations -I G3D9/build/osx-i386-g++4.2/include/
  -I /usr/local/include/ -I /usr/include/ -o build/0-Cubes
  -Wl,-w -arch i686 -msse3 -mfpmath=sse
  -Wl,-headerpad_max_install_names -L G3D9/build/osx-i386-g++4.2/lib/
  -L/usr/local/lib/ -L/usr/lib/ -framework AGL -framework
  IOKit -lGLG3Dd -lavformat -lavcodec -lavutil -lG3Dd -lzip
  -framework Cocoa -framework Carbon -lz -framework OpenGL
  -lpthread -ljpeg -lpng -multiply_defined suppress
  -all_load source/App.cpp source/Scene.cpp
```

So instead of typing that directly, you're going to use a script that produces the command line for you. The script is called **iCompile** and it comes with G3D. It is written in Python and you are welcome to look at the source code for it. For now all that you need to know is that if you type:

```
icompile
```

in the directory containing your project, it will figure out the appropriate g++ command line and execute it. You can use the `--verbosity 2` command line option if you'd like to see the underlying commands that are being executed. The first time you run iCompile on a project it will ask you to confirm that you really want to compile. Press "Y".

11. To see a complete list of icompile options, run

```
icompile --help
```

You will use the `--opt`, `--run`, `--doc`, `--gdb`, and `--clean` ones frequently.

Tip: I recommend that you always work from a single, persistent Emacs instance. This will keep you from accidentally opening the same file in two different sessions. It will reduce your development time. You can keep your hands on the keyboard while compiling, and can cut and paste between files and between code and the shell using only Emacs keyboard commands. It will also reduce the overhead of editing. I've seen students who opened a source file, found the line they needed to change, edited it, closed the editor, and then compiled. The compiler would report an error on the very next line, so they re-opened the same file, searched for the line, etc...it took those students more than twice as long to debug a program as the ones who simply kept their files open and on the right line.

4.2 Graphics Programming with G3D::GApp

1. Move `main.cpp` to `source/main.cpp`. Edit your `main.cpp` to look like:

```
#include "App.h"

// Tells C++ to invoke command-line main() function even
// on OS X and Windows.
G3D_START_AT_MAIN();

int main(int argc, const char* argv[]) {
    GApp::Settings settings(argc, argv);
    settings.window.width       = 1440;
    settings.window.height      = 800;

    return App(settings).run();
}
```

Note that you can't recompile because you haven't written the new classes that are being referenced yet.

2. **Add `main.cpp` to Subversion**. Whenever you create a new file, it is a good idea to add it to revision control right away so that you don't later forget. Execute:

```
svn add source
```

This command will *mark* `source/` and `source/main.cpp` for addition to your repository. You can see this by running `svn status`. They haven't actually been added yet. To do that, commit your changes with:

```
svn commit -m "Added main.cpp"
```

Now your file is on the server and safe from local changes. If you modify the file, you will need to commit the new version, but never need to add this file again.

3. **Create `source/App.h`**.

C++ splits code into header and implementation files. By convention, we put one class in each header. Header files describe the interfaces to classes and functions. They include both public and private data because the compiler needs to know the size of each class, and the private data affects the size. Write a `App.h` header. This that contains the interface for the `App` class that will manage the graphical user interface (GUI) and general 3D scene state for your program. It should look like:

```
#ifndef App_h
#define App_h

#include <G3D/G3DAll.h>
```

Tip: Save frequently and whenever you compile or switch buffers. This will keep you from accidentally compiling out-of-date code and will increase the chance of recovering your program in the event of a crash. Graphics programs interact with the OS at a low level and can crash your computer.

Tip: You don't have to list your method arguments vertically. I just did that here so that the lines would fit on the page in the PDF. If you do make them vertical, it is easier to read if you line them up in columns.

```
class App : public GApp {
private:

public:

    App
      (const GApp::Settings&  settings);

    virtual void onInit();

    virtual void onPose
      (Array<Surface::Ref>&   surface3D,
       Array<Surface2D::Ref>& surface2D);

    virtual void onGraphics3D
      (RenderDevice*          rd,
       Array<Surface::Ref>&   surface3D);
};

#endif
```

The preprocessor commands at the top of the header are called a **header guard**. They are a common trick used to ensure that this header is never included twice into your program, since doing so could cause hard-to-debug compile time errors.

The include preprocessor command imports the definition of the G3D library. The C++ language provides only computation, not routines for managing the GUI, communicating with the graphics card, or even basic file I/O. All of that is contained within libraries. We're going to use the G3D library as a common and platform-independent source of utility routines. It is good for learning 3D graphics because it resembles a film or game rendering engine, but exposes most of its functionality so that you can replace parts with your own code.

The `App` class inherits from `GApp`, which is part of G3D. Look it up in the G3D documentation (be careful to use the version 9.00 beta documentation on our server and not the older 8.00 version on SourceForge). `GApp` provides a number of event handlers (a.k.a. callbacks), which are implemented as virtual methods. We can override these to respond to specific events. In this project we're going to execute some code on initialization, when the scene is "posed" for rendering, and when the scene is rendered in 3D.

Tip: Forgetting the semicolon at the end of the class definition, forgetting the `#endif`, and incorrectly copying the base class's method signatures when overriding them are common bugs that create misleading compiler messages.

4. **Add `App.h` to revision control.**

```
svn add source/App.h
svn commit -m "Added App.h"
```

From here on, I'm going to assume that you add every file that you create without needed explicit instructions. Take care to not add generated files (e.g., the `build` directory, Emacs backup files ending in tilde) to the repository. If you accidentally add something, you can `svn revert` that file. See the Subversion manual and the `svn --help` command for detailed instructions.

5. **Create `source/App.cpp`** to implement your `App` class by typing in the following:

```
#include "App.h"

App::App(const GApp::Settings& settings) : GApp(settings) {
}


void App::onInit() {
   // Put initialization code here
}


void App::onPose
 (Array<Surface::Ref>&   surface3D,
  Array<Surface2D::Ref>& surface2D) {

     (void)surface3D;
     (void)surface2D;
}


void App::onGraphics3D
  (RenderDevice*         rd,
   Array<Surface::Ref>&  surface3D) {

     (void)surface3D;
     Draw::axes(CoordinateFrame(), rd);
}
```

All of the `(void)` expressions are just a way of telling the compiler that you're intentionally ignoring the value of another expression. In this case they serve to prevent the compiler from warning you that you ignored the parameters to most of the methods.

Tip: You should always investigate warnings, and modify code to avoid them in cases where you verify that there is no problem. That way you will notice the new warnings if introduce incorrect code later.

The only interesting thing in this class is the `App::onGraphics3D` method, which uses the `G3D::Draw` utility class to render the default coordinate frame as a set of arrows. Those axes will help us stay oriented as we create a more interesting scene.

6. **Run it!** Compile and run your program using iCompile. You should see a set of colored axes on a blue background and some additional debugging tools that G3D adds to every program. You can disable those debugging tools later in your `App::onInit` method, but for simplicity just leave them there right now.

   By default, `G3D::GApp` creates a `G3D::FirstPersonManipulator` that allows you to move the 3D camera. This manipulator uses common first-person

PC video game controls. The 'W', 'A', 'S', and 'D' keys on the keyboard will translate the camera forward, left, back, and right relative to its own axes. If you press the right mouse button (or press Shift and the mouse button for a single-button mouse under OS X), the mouse rotates the yaw and pitch of the camera. It requires you to press a button because otherwise using the mouse with the GUI would also move your viewpoint. G3D contains other manipulators with different control styles, and you can write your own or use none at all. This is only the default. Move the camera around a bit to get a feel for the controls, and then exit the program.

4.3 One Cube, and the Posing Design Pattern

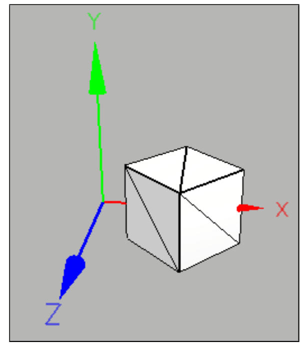

Figure 3: *A scene with one instance of `cube.ifs`, and a set of axes for debugging.*

We begin by building the simple scene containing a single cube lit by an infinitesimally small (i.e., point) light source shown in Figure 3. The cube will be centered 1 m along the positive x-axis and rotated 45 degrees about the vertical axis.

There are more convenient ways of creating the objects described in this section, and fairly helpful defaults for all of the values. I'm using a verbose initialization process here to make clear what options you can change. In the G3D documentation you can find details about these settings and even more options.

The concept of reducing a complex model to just the information needed to render a frame is common in computer graphics. "**Pose**" is the name that I give this process; there is no universally accepted term for it. In the G3D API, a "**surface**" is the boundary of a 3D object. That is, what you would call a surface in everyday life. Beware that for historical reasons, under some graphics APIs, "surface" also a name for the image that is being rendered.

1. **Create a lighting environment**. First, we create the environment around the box. This consists of the lighting and a "sky box" that is an infinite cube painted with distant objects so that we appear to be in a large environment. Declare member variables `m_skyBoxTexture` of type `Texture::Ref`, `m_skyBoxConstant` of type float, and `m_lighting` of type `Lighting::Ref`.

   In `App::onInit`, initialize these members as follows:

```
Texture::Specification skyBoxSpec;
skyBoxSpec.filename       =
    System::findDataFile("cubemap/whiteroom/whiteroom_*.png");

skyBoxSpec.desiredFormat = ImageFormat::RGB8();
skyBoxSpec.dimension     = Texture::DIM_CUBE_MAP;
skyBoxSpec.settings      = Texture::Settings::cubeMap();
skyBoxSpec.preprocess    = Texture::Preprocess::gamma(2.1f);

Lighting::Specification lightingSpec;
lightingSpec.lightArray.append
    (GLight::point(Point3(7, 10, 4), Power3::white() * 100.0f));

lightingSpec.environmentMapTexture = skyBoxSpec;

m_skyBoxTexture = Texture::create(skyBoxSpec);
m_skyBoxConstant = 1.0f;
m_lighting = Lighting::create(lightingSpec);
```

   The "specification" classes are a way of setting a complex set of arguments to the factory methods. This is a design pattern that G3D uses for most major classes. It isn't the only way of handling complex initialization arguments, but it is one I've come to prefer (you'll see some of its advantages in the next project). I think the best way to teach design patterns is to have you just start using them. You'll pick up a lot of small programming tricks like this throughout the course that will be new tools you can later apply to other problems.

2. **Create a geometric model of a cube.** Declare a member variable `m_cubeModel` of type `ArticulatedModel::Ref`. Initialize it with:

```
ArticulatedModel::Specification modelSpec;
modelSpec.filename = System::findDataFile("cube.ifs");

modelSpec.preprocess.xform = Matrix4::scale(1.0f, 1.0f, 1.0f);
modelSpec.preprocess.setMaterialOverride(Color3(1.0f, 1.0f, 1.0f));

m_cubeModel = ArticulatedModel::create(modelSpec);
```

3. **Pose the cube.** Your cube **model** is centered at the origin and aligned with the axes. To render it at the desired location, you need to **pose** it. Posing places a model at a specific location and reduces it to just the information necessary for rendering, which for our application infrastructure is an array of **surfaces**.

   The code to pose the cube 1 m along the x-axis and rotate it 45 degrees about the y-axis in the `App::onPose` method follows.

```
void App::onPose
(Array<Surface::Ref>&   surfaceArray,
 Array<Surface2D::Ref>& surface2D) {

    (void)surface2D;
    m_cubeModel->pose(surfaceArray, CFrame::fromXYZYPRDegrees(1,0,0, 45,0,0));
}
```

   The word `xform` is a common graphics abbreviation of "transformation." That is the name of the variable, so you have to use it. `CFrame` is an abbreviation of `CoordinateFrame`. That is an alias (typedef), so it is optional if you prefer to type out really long things.

4. **Send the geometry to the graphics card.** Most modern computers contain a general purpose CPU and a dedicated graphics processor, which is also known as a GPU or graphics card. Part of the graphics card's memory is dedicated to a data structure called a **framebuffer**, which is essentially an image of what should be displayed on the screen. A circuit on the graphics card continually sends the framebuffer to the display. To display an image, we therefore need to transfer information from the CPU to the GPU. We'll let the G3D library handle most of this for our first program. Add the following code to `App::onGraphics3D`:

```
Draw::skyBox(rd, m_skyBoxTexture, m_skyBoxConstant);

// Draw the surfaces, with appropriate lighting
Surface::sortAndRender(rd, defaultCamera, surface3D, m_lighting);

// Draw the surfaces again in wireframe mode
// so that we can see the mesh.
```

```
rd->pushState();
{
    rd->setRenderMode(RenderDevice::RENDER_WIREFRAME);
    rd->setLineWidth(2);
    rd->setColor(Color3::black());
    Surface::sendGeometry(rd, surface3D);
}
rd->popState();

// Visualize the light sources to help with debugging
Draw::lighting(m_lighting, rd);
```

In C++, braces have three purposes: they create a local scope, they group a set of statements into a single statement, and they incidentally trigger indenting in the editor. Here we're using them for the trivial purpose of creating indenting. The indenting helps us to remember to call `rd->popState`.

There are a lot of things going on in this code that I did not attempt to explain. What algorithm does the graphics card use to draw the scene? How much of that algorithm is in the G3D library, how much is in the OpenGL API that G3D uses to communicate with the graphics card, how much is in the graphics processor? What are the formats of the cube and sky data files? Why does the image look like "computer graphics" instead of like a photograph? These are the kinds of questions we'll spend the rest of the semester investigating.

4.4 A Scene Data Structure and the Model/Entity Design Pattern

To create a scene with more than one cube we need some data structure that abstracts over the parts. For this project, we use the simplest solution: a class comprising the various lighting constants and an array of array of objects. We'll see more sophisticated scene data structures soon, but currently have no motivation for anything more structured than an array.

We also need a better abstraction of the difference between a model and an instance of a model. The Model/Entity design pattern stores the geometric template common to a class of objects in a **model** and the information about a particular instance in an **entity**. Under this pattern, for example, A scene containing thousands of trees might be represented by a single tree model and many entities that each reference that shared model and store a unique location and orientation.

1. **Create the `Entity` class**. We can use `G3D::ArticulatedModel` class as our model class, but you need to create the `Entity` class. A functioning header and implementation follow.

```
/**
  \file Entity.h
  \author Morgan McGuire, morgan@cs.williams.edu
 */
#ifndef Entity_h
#define Entity_h

#include <G3D/G3DAll.h>

/**
  \brief An instance of an object in the world.
  Contains a single rigid-body frame and never
  moves from its initial position.
 */
class Entity : public ReferenceCountedObject {
public:

    typedef ReferenceCountedPointer<Entity> Ref;

private:

    CFrame                          m_cframe;
    ArticulatedModel::Ref           m_model;

    /** Called from Entity::create() */
    Entity
     (const CFrame&                 cframe,
      const ArticulatedModel::Ref&  model);

public:

    /** \brief Creates  new Entity. */
    static Ref create
     (const CFrame&                 cframe,
      const ArticulatedModel::Ref&  model);

    /** \brief Appends the surfaces of this entity to \a surfaceArray.
        Called from App::onPose(). */
    void onPose
     (Array<Surface::Ref>&          surfaceArray) const;
};

#endif
```

(CFrame is a shorthand for CoordinateFrame in G3D. You can use them interchangeably).

```
#include "Entity.h"

Entity::Entity
  (const CFrame&                  cframe,
   const ArticulatedModel::Ref&   model) :

    m_cframe(cframe),
    m_model(model) {
}


Entity::Ref Entity::create
  (const CFrame&                  cframe,
   const ArticulatedModel::Ref&   model) {

    return new Entity(cframe, model);
}


void Entity::onPose(Array<Surface::Ref>& surfaceArray) const {
    m_model->pose(surfaceArray, m_cframe);
}
```

Note the C++ colon-syntax for initializing member variables in the constructor. Think of this as invoking the constructors of the members. You'll get warnings if you don't initialize them in the same order that you declared them (...and I don't want to see warnings when I compile your code.)

2. **Create the `Scene` class**. You've seen an example of a reference-counted C++ class (`Entity`), and know how to declare and initialize a lighting environment and objects. Apply that knowledge by creating a `Scene` class. For this class:

   (a) Make all of the members private and use accessor methods for any that you require access to from `App`.

   (b) Pass a `const std::string& name` argument to the factory method. This will be the name of the scene. For the moment, ignore the name and always create the single-box scene.

Tip: Read the explanation of the `Ref` pointer type and `create()` factory methods in the C++ section of the *Tools* document.

3. **Test your `Scene` class**. Your single-box scene should look the same as before, but there should be significantly less code in the `App` implementation now.

4.5 The Cornell Box

The **Cornell Box** is a real-world box at Cornell University that has been long used for photorealistic rendering experiments. The idea is that by constructing a real scene containing only well-measured geometric primitives, we can create a perfect virtual replica and then measure rendered results against real photographs. There have been many variations on the Cornell Box. We'll model the specific one shown

Tip: Look up G3D's debugging routines, especially `debugPrintf` and `debugAssert`.

in Figure 4, and estimate the geometry rather than working from measurements (N.B. the measurements are available on the Cornell web page.)

Figure 4: *Photographic reference of the real Cornell Box from* `http://www.graphics.cornell.edu/online/box/compare.html`.

This Cornell Box can be modeled using seven instances of rotated, translated, and scaled cubes. When creating your Cornell Box, scale the *models*, but don't rotate and scale them. Instead rotate and translate the *entity* placement. Were we animating the scene that design would give us more intuitive control of the objects. It also lets us reuse objects. For example, all three white walls should be different entities that use the same model.

1. **Specify the scene**. In `App::onInit`, pass `m_settings.argArray[1]` to the `Scene::create` factory method. This will allow us to select the scene to load from the command line. For example, executing

   ```
   icompile --run Cornell
   ```

   will pass the string `"Cornell"` to your `Scene::create` method. (Consider creating a default value or error message so that your program doesn't crash mysteriously if you forget the argument!) By the end of this project, you need to support three arguments: "cube", "Cornell", and whatever single-word name you give your custom scene (discussed later).

Tip: Print the scene name when you load it; this helps catch typos on the command line.

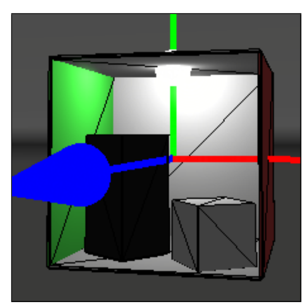

Figure 5: *A rough approximation of the Cornell Box model using seven instances of* `cube.ifs`.

2. **Model the Cornell Box scene.** Write code to create the `Entity`s and `ArticulatedModel`s for the Cornell Box. Don't forget to place the light and dim the background. We don't know how to model the "color" of a mirror yet, so set the mirrored box to be black. You can skip the black card near the ceiling. That was there in the original experiment to avoid lens flare from directly imaging the light source, since lens flare was not the point of the experiment.

   You can chose the scale and need not worry about the precise colors and angles.

Ensure that the walls have nonzero thickness. I chose $1\ \mathrm{mm}$ walls for a $1\ \mathrm{m}^3$ box.

4.6 A Custom Scene

The single cube was my example to show the parameters you can adjust and how to initialize certain classes. The Cornell Box is a classic rendering test that shows me that you have sufficient control of the classes to model a given scene. For any rendering project you'd probably make simple scenes like this as initial targeted experiments. Then you'd make a more visually compelling scene to demonstrate that your implementation scales to the complexity of more interesting data sets.

Design a visually impressive scene of your own and model it using cubes, lights, and a sky box. For example, decided to create the dog shown in Figure 1 (*you should not make the dog–you should make something else*.) I'm expecting something of about the complexity of my dog. Although you're welcome to go beyond that if you enjoy the process, I'm not expecting the Taj Mahal for Project 0; it just has to be more interesting than the Cornell Box!

Tip: Press F4 to take a screenshot and F6 to record video in any G3D program.

If you're stumped for artistic inspiration, note that legos, Lincoln logs, and most other building toys, let alone most houses and other buildings are just scaled cubes...

5 The Gallery

Each week I'll collect everyone's images and put them on a web page (without names), so that we can see each other's work. I'll show that page in lecture as well. This is a common practice in art classes. It gives everyone a sense of the standard of the class and presents new ideas. It is also nice to see the final products of the projects that you collectively worked on.

But this is not an art course. So why do I require a "visually impressive" image in every project? Visual communication and presenting your work effectively are important in any field. Learning how to compose images that read clearly, with good color palettes, camera positions, overlap, and lines is a valuable skill, and one that anyone can acquire with practice. We'll incidentally explore composition in the context of the images we see throughout the semester in lecture.

In computer graphics in particular, it is important to leverage visual communication skills to present algorithms in a compelling way. On one hand, we'd like like algorithms to be judged by quantitative results and analysis. On the other hand, following such analysis is a large investment on the part of the audience, and a single image can prove that an algorithm is indeed sufficient for a task. As an audience member, if someone can't show you a picture demonstrating that his or her algorithm does what you want it to, why would you bother following an analysis of just how poorly suited it is?

Most computer graphics papers and talks therefore begin with a single, visually compelling image, often called a teaser. If the teaser grabs you, then you will investigate the rest of the work to see how well the technique applies under specific targeted experiments. Those targeted experiments isolate a single phenomenon and explore how parameters and specific input scenarios affect it. They typically employ common datasets to allow comparison with previous techniques, the results of which are often shown side-by-side. Take a look at X and Y.

The same process is also applied outside of pure research in the context of production and engineering. Say that a technical director at a film company is investigating new shadowing methods. He or she would render a few scenes from that company's previous film with the new method to show everyone what to expect from the new algorithms. He or she would then make specific images to investigate the algorithm more carefully. For example, the hard shadow of a single edge under a point light, the soft shadow of that edge under an area light, shadows from translucent objects, cast by and on curved surfaces and so on.

All of these images are **results**, which has a formal meaning in this context. The process of creating result image must be repeatable and clearly explained. Unless that is explicitly part of the technique, result images should not be retouched in tools like Photoshop–the pixels displayed must be the ones that come out of the program. There are some gray areas of retouching: cropping and gamma correction for presentation are probably acceptable in most cases; scaling and color adjustment should probably be explained. For targeted experiments, the experimenter should seek to produce a representative image, a best case and a worst case so as to accurately describe the expected behavior.

References

COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. 2008. Subversion complete reference. In *Version Control with Subversion*. O'Reilly, ch. 9. http://svnbook.red-bean.com/en/1.5/svn.ref.html.

ROBERTS, A., 2009. Getting to grips with Latex - Mathematics, December. http://www.andy-roberts.net/misc/latex/latextutorial9.html and http://www.andy-roberts.net/misc/latex/latextutorial10.html.

VAN HEESCH, D., 2010. Doxygen 1.7.1 manual. http://www.stack.nl/~dimitri/doxygen/manual.html.

Index