CS 371 Project 0:

Cubes

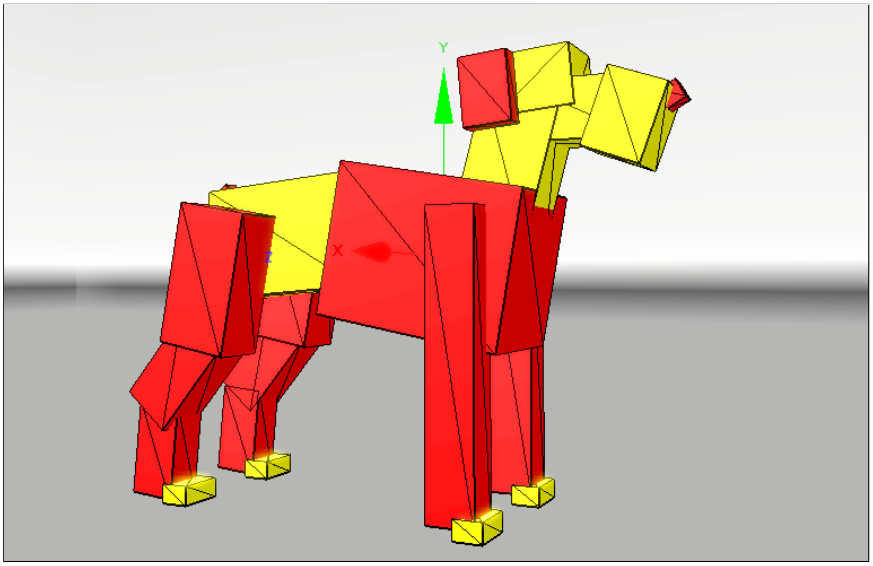

Figure 1: *A dog modeled with translated, rotated, and scaled cubes. By the end of this project you'll know how to create scenes like this programmatically and write an interactive real-time 3D renderer for viewing them.*

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, make an interesting scene data file for it, 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 reasons, the handout is 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 structuring 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/Surface design pattern
2. **Some CS371 software development tools:**
   (a) The C++ programming language
   (b) The Subversion (svn) revision control system
   (c) The G3D library and iCompile script
   (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:	Thursday,	September 6	
Due:	Monday,	September 10,	12:00 pm (noon)

This is an easy, individual project. Most projects in CS371 will be more challenging and completed as part of an assigned team.

This warmup project is structured slightly differently than the other projects this semester. For most projects, you'll start working on Tuesday and have a graded checkpoint worth 50% of the points on Thursday. We'll then tackle the crux of the project together in lab on Thursday and you'll have until the following Monday to finish.

For this project, don't start before the scheduled lab session. We'll begin it as a class in lab on Wednesday, September 6th. 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 25 statements and 204 comment lines (as reported by iCompile), including the Doxygen comments that generate the report. Most of the code is given to you in this handout.

Track how much time you spend on this project. 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 rely mostly on these three documents when evaluat-

Tip: I grade against the Specification, which is usually terse. Advice and Walkthrough sections in the project documents help you form a plan for satisfying the specification.

ing your work. I will only review small sections of your code and may not run your program at all when grading. This is a 300-level course, which means that I'm evaluating how well you can describe and reason about algorithms and software. Although I will offer programming advice, I assume that you already know how to program quite well; making a program is no longer an explicit goal of an assignment at this level.

Tip: Use your time wisely. Decide up front how much time you are going to spend on each part of the specification, and move on or seek help if something starts taking too long.

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. Structure a directory in svn with exactly the following subdirectories, some of which may be empty for this project:

   (a) `data-files` - *Files needed for running your program*

   (b) `source` - *Your .cpp and .h source code*

   (c) `doc-files` - *Files needed when viewing your documentation and report*

   (d) `journal` - *Empty for now, but we'll use this next week*

   (iCompile will create a `build` directory as well. Do not add it to svn)

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

3. Create the following scenes in human-readable `.scn.any` data files, using only the `models/cube/cube.obj` and the `cubemap/whiteroom/whiteroom-*.png` environment map files:

   (a) A single, white 1 m^3 cube rotated 45 degrees about the vertical axis, with center at x=0m, y=0m, z=-2m.

   (b) A model of the Cornell Box that is pictured in Figure 6.

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

4. Create overview and entry point (i.e., method, class, function, variable) documentation for your software using Doxygen.

5. Create the report described in Section 2.1.

As with all projects, you may negotiate changes to this specification with me if you discover a more effective way for you to achieve the learning objectives of the project. For example, students often propose loosening the restrictions on point 3c in this project.

2.1 Report

Create your report in the file `mainpage.dox`. The entire source file will be a single giant comment that contains within it text and markup. The actual readable HTML report is generated in `build/doc` when you type `icompile --doc` at the command line. Note that the walkthrough at the end of this document helps you to set up your first report.

Figure 2: *What is your graphics vision?*

1. **Vision.** You're going to learn a lot about numerical methods, software engineering, hardware, geometry, and specific algorithms and data structures in this course. But that's not what drives you. In your head, you have a vision of something created with computer graphics. Maybe it is the vaulted arches of an immense cathedral, an epic space battle in silver and white, the mist-filled caverns of a 3D fractal, or two cartoon characters exploring a jungle ruin. Your vision can be a single scene, an animation that tells a story, a game, or a complex interaction. **Articulate your personal vision in a section at the start of your report, using text and images**. View every project as a way to use computation to realize your vision. I'll show many 3D films and research results designed to inspire or enhance your vision, and a (graded) requirement of each project is a compelling image, which is the primary place where you will explore your realization. You are welcome to articulate a new, personal driving vision at any point in the course, and to experiment with different ideas on different projects.

2. **System Overview** 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. Feel free to use lists or tables.

3. **Coordinate System.** 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.

4. **Results.** Include images of the single scene, the Cornell Box, and your custom scene. Put the actual image files in the `doc-files` directory and link thumbnails using the `\thumbnail` and `\video` Doxygen commands. Especially for your custom scene, you'll have spent a while making it. Reap the benefit of that time investment by making lots of images or a few videos, not just one. I'll show some of these images in class next week so that we can see what each other created.

5. **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 **share strategies** for finding the answers.

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?

(a) What are the differences between the `Scene*` and `shared_ptr<Scene>` 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.obj" stored on the file system? What made the scene data file look there?

6. **Time.** We study software engineering techniques in CS371, and your of our goals will be learning ways to work as an effective team member and as an efficient individual programmer. I track the class average time for each project and compare it to some baselines in lecture each week as one way of measuring productivity. Your grade isn't affected by how long you spent on the project, but you do receive points for answering these questions.

   (a) How many hours you spent on this project on **required** elements, i.e., the minimum needed to satisfy the specification. Include the time that you spent in the scheduled lab.

   (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.

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.** In the extreme, for many projects you can receive a favorable evaluation even if your program does not compile or execute as long as you document it well and write a good report. So, invest time communicating clearly in your report and documentation; that's what I see when grading, not the time that you spend in lab or the insights hidden in your head.

Later parts of the specification (e.g., the report) are usually worth more points than those at the beginning (e.g., your initial code). In light of this, I recommend working backwards for your first draft of the project. Begin each project by creating and document your major code entry points, and write the report with some placeholder images and numbers. Do this before you ever write a line of executable code. Doing so leaves you with a viable submission after the first hour of work...from then on, you're just increasing the value of your submission. This way, you don't end up writing the report and documentation at the last minute. This process also helps you organize your program design. The alternative bottom-up approach makes no sense for projects of this scale: why would you write helper functions or implement method bodies when you don't know what they are helping or what the method does? You'll also find that, when we move on to team projects next week, it is essential that the team agrees on the program and report structure before beginning work.

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 compelling" 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. 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 3: *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 3.

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 or create `~/.local_bashrc` in your favorite editor (mine is Emacs) and ensure that your environment variables contain the CS371 paths. These should look 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 editing environment**. Open `~/.emacs` and add these lines:

```
(setq c-basic-offset 4) ; 4-space indenting

; Compilation shell: M-x cshell
; Makes error line numbers into emacs links
(defun cshell ()
  (interactive)
  (shell)
  (compilation-shell-minor-mode)
)

;; make a shortcut for the goto-line function
(global-set-key [f8] 'goto-line)
```

5. **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 (note that most of this line is already in the file, and be careful not to use # anywhere regardless of what is already present, since that is the comment symbol for this configuration script). 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.

6. **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 /Volumes/scratch
```

7. **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.

8. **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.

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.

```
#include <stdio.h>

void f() {
     throw "Exception";
}

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

9. **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:

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.

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

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

10. **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 function `f`'s 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".

    (h) Fix your program by commenting out the call to `f()` again, and save `main.cpp`.

11. **Compile with iCompile.** You could continue to directly invoke g++ for the rest of the semester, 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
```

I don't want to type that–I don't even want to see it again. So, instead of running g++ 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. You do, so press "Y".

12. To see a complete list of iCompile options, run

```
icompile --help
```

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

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.

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.

Tip: Always work from a single, persistent Emacs instance. It can have lots of different files and multiple command-line shells open. Do not open up a second copy of Emacs, ever. 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.

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. That file 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>

class App : public GApp {
private:

public:

    App
      (const GApp::Settings&  settings);

    virtual void onInit();

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

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

#endif
```

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.

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.

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.

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 at

http://graphics.cs.williams.edu/courses/cs371/f12/G3D/manual and

not the older 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.

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.

Beware that the files you add consume space on the single, shared disk used for all students in CS371. Please don't check in huge files–limit yourself to small amounts of data and let me know if you have an unusual, reasonable need to commit a large file. There's a limit of about 2 MB per file that is enforced by the system, but this is mostly to help you catch accidental file additions. You could easily thwart it by checking in lots of small files, for example. You're responsible for being considerate about your own file usage.

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<shared_ptr<Surface> >&   surface3D,
  Array<shared_ptr<Surface2D> >& surface2D) {

    (void)surface3D;
    (void)surface2D;
}


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

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

Allof 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.

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.

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.

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 "control" 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 by pressing the "esc" key.

4.3 Draft Report

You should write your report first. Otherwise you'll end up scrambling at the last minute and write a weak report, which is a poor strategy since it is worth as much as 50% of your grade. So, although you haven't yet written a program that satisfies the specification, start the report right now.

1. **Create mainpage.dox**. This file resides in the root of your project (above `source`). The name or location of the file doesn't actually matter, but it will be easier for us to find each other's reports when collaborating if we follow a convention. This file is technically a C++ file, but it contains no executable code. Instead, it is a giant comment written with Doxygen markup, which is a mixture of LaTeX and HTML. Start your contents out like this:

```
// -*- html -*-
/** \file mainpage.dox \mainpage

\section Report
<b>Project</b>: 0-cubes
<br><b>Team</b>: Morgan McGuire (morgan@cs.williams.edu)
<br><b>Date</b>: September 10, 2012

\subsection Vision Vision

My driving vision for 3D graphics is...

\subsection Overview Overview

The program begins in main(), which then creates an
instance of App on the stack...

\subsection coords Coordinate System

\thumbnail{temp1.jpg, The 2D Coordinate System}
\thumbnail{temp2.jpg, The 3D Coordinate System}

\subsection Results Results

\thumbnail{temp1.jpg, The white cube}
\thumbnail{temp2.jpg, The Cornell Box}

\subsubsection scene My Scene

\thumbnail{temp2.jpg, Main view with wireframe}
\thumbnail{temp2.jpg, Main view without wireframe}
\thumbnail{temp2.jpg, My scene from above}
\thumbnail{temp2.jpg, Another view}

\subsection Questions

<ol>
   <li> <b>Scene* vs. shared_ptr&lt;Scene&gt;:</b>
      The Scene* syntax ...</li>

   <li> <b>ICE_EXTRA_SOURCE:</b>
     This environment variable ...</li>
</ol>

\subsection Time

<b>Time on required elements:</b> ?? hours
<br><b>Time on optional elements:</b> ?? hours

*/
```

Figure 4: *The draft report with placeholder images and text.*

Note that I was able to write a lot of the report without knowing anything about the project. Also note that I took some care to make it look attractive and clearly organized. I'm evaluating you partly on clarity of presentation, so proper grammar, spelling, a clear layout, and effective use of images and diagrams earn you points (and after all, this class is *about* visual communication!)

The `thumbnail` commands insert references to images stored in the `doc-files` or `journal` directory. Since you don't have any results, insert some placeholder images for now.

2. **Generate the HTML**. Run `icompile --doc` to generate `build/doc/index.html`, which is your report. Note that iCompile generated a file name `Doxyfile`, which controls some aspects of this process. You should check this in to svn. You should also modify the project name appropriately.

   Your report should initially look something like figure 4.

3. **Articulate your vision**. Write about your driving vision. Don't hesitate to use images here if some examples would help. Keep this to one paragraph (or even

one very good sentence).

4. **Coordinate system**. Answer the coordinate system question now. You'll need to know this information shortly to complete the next set of tasks, so you might as well lock in the points for looking it up now as well. This is not a coincidence, of course–I try to ask questions in the report that you should be asking yourself anyway to complete the assignment effectively. This is another reason to start the report early.

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

5. **Questions**. Sketch out answers to the four explicit questions on the report now.

4.4 One Cube

We begin by building the simple scene containing a single cube lit by an infinitesimally small (i.e., point) light source shown in Figure 5. The cube will be centered 1 m along the positive x-axis and rotated 45 degrees about the vertical axis.

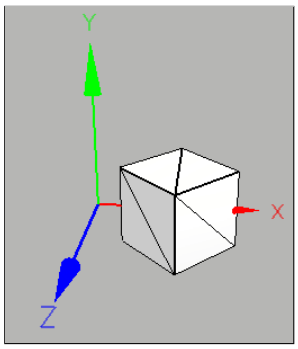

Figure 5: *A scene with one instance of* `cube.obj`*, and a set of axes for debugging.*

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. **Check arguments in `main`.** We're about to pass the name of the scene to load to the program as a command line argument. This means that we should print an error if the user (i.e., you) forgets the argument rather than letting the program crash. Modify your `main` function to test that there are exactly two values in `argv` with:

```
if (argc != 2) {
    printf("Must specify the name of the scene\n");
    exit(-1);
}
```

2. **Add a scene.** The `G3D::Scene` class provides all of the code needed to read data files describing 3D scenes. Add one to your program by adding a `shared_ptr<Scene> m_scene` protected member variable in `App.h`. Initialize it in `App::onInit` as:

```
m_scene = Scene::create(m_settings.argArray[1]);
```

3. **Create a data file**. Create a `data-files` directory in the root of your project (i.e., next to `source`). In that directory, create a file named `cube.scn.any` that contains:

```
// -*- c++ -*-
{
    name = "Cube",

    models = {
        cubeModel = ArticulatedModel::Specification {
            filename = "models/cube/cube.obj";
            stripMaterials = true;
            preprocess = (
               setMaterial(all(), all(), Color3(1, 1, 0));
            );
        };
    };

    skybox = {
        texture  = "cubemap/whiteroom/whiteroom-*.png";
    };

    entities = {
        light0 = Light {
            type     = "SPOT";
            position = CFrame::fromXYZYPRDegrees(10, 10, 10, 45, -35, 0);
            spotHalfAngleDegrees = 30;
            power    = Power3(6000,6000,6000);
        };

        cube0 = VisibleEntity {
            model = "cubeModel";
            position = CFrame::fromXYZYPRDegrees( 0, 0, 0, 0, 0, 0);
        };

        camera = Camera {
            position = CFrame::fromXYZYPRDegrees( 0, 0, 5);
        };
    };
}
```

This creates a simple scene graph. It contains one kind of model, which in this case is named "cubeModel". It then creates one entity called "cube0" that uses cubeModel for its geometry, one light, and cone camera. The separation of an instance of an object from its template allows many objects in the scene to share an appearance without having to store redundant appearance data in memory. This is the Entity/Model data structure.

This data file looks a lot like C++ code, although it is not. This allows Emacs to properly indent and syntax-color the file (after you've saved and reloaded it). Most of the capitalized names in the file correspond to G3D classes, and

this is just an easy way of instantiating a lot of classes without hardcoding the information into your actual program. You can check the G3D documentation to see some of the options to these classes.

The "specification" classes are a way of setting a complex set of arguments to factory methods and constructors. 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.

4. **Test**. Run your program, providing the **name** of your scene, not the filename (i.e., "Cube", not "cube.scn.any") on the command line with:

   `icompile --run Cube`

   If anything goes wrong, run under the program gdb with `icompile --gdb Cube` in order to debug. Note that your program will still just draw the axes.

5. **Pose and render the scene.** Pose the scene by altering `App::onPose` to call the super-class method and then the scene's pose method:

```
void App::onPose
(Array<shared_ptr<Surface> >&   surface3D,
 Array<shared_ptr<Surface2D> >& surface2D) {
    GApp::onPose(surface3D, surface2D);

    m_scene->onPose(surface3D);
}
```

   You don't yet know how to draw the cube, so we'll rely on some built-in G3D code to provide a preview of the scene. You'll spend the first half of the semester writing a much better quality–but slower–renderer yourself, and then the last half of the semester writing a fast and reasonable-quality renderer. For now, the magic incantation that you need is just:

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

    defaultRender(rd, defaultCamera, m_scene->lighting(), surface3D);
    Surface::renderWireframe(rd, surface3D);

    Draw::axes(CoordinateFrame(), rd);
    Draw::lighting(m_scene->lighting(), rd);
}
```

   Run your program again and you should be able to see the cube.

6. **Model the scene.** The specification requires you to make a white cube at a particular position. I gave you the code to create a yellow cube at the origin. Modify the Color3 argument for the model to change the color (it expects red, green, and blue values–see G3D::Color3 documentation). Modify the CFrame::fromXYZYPRDegrees arguments for the entity to position the cube. That method name stands for "x, y, z, yaw, pitch, roll"; see the documentation in G3D for more details.

7. **Update the report.** You just did a lot of work to create and render the cube...and received 0 points for it. That's because you haven't updated your report yet. If I graded your project at this stage, I wouldn't know that you completed the cube because there is only a placeholder in your results section. Take a picture of the cube scene and add it to the report. You may want to crop the image in photoshop so that there isn't too much empty space around the sides.

   Now you've locked in the points for completing the cube scene from the specification. From here on, I'm assuming that you're updating the report after each major step.

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 in Figure 6, and estimate the geometry rather than working from measurements.

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

Figure 6: *The Cornell Box.*

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 translate 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. **Create a new data file**. Create a `cornell.scn.any` file that will be your Cornell Box. Remember to change the "name" field inside of the file.

2. **Model the Cornell Box scene.** To do this, you'll need to be able to distort the shape of the cube to make arbitrary rectangular slabs. Just like `setMaterial`, there are program-like commands that you can put in a scene file to modify geometry. These are listed in the `G3D::ArticulatedModel::Specification` constructor documentation. In this case, you want to use the `transformGeometry` command, which might make part of your data file look like:

```
squishedCube = ArticulatedModel::Specification {
  filename = "models/cube/cube.obj";
  stripMaterials = true;
  preprocess = (
    setMaterial(all(), all(), Color3(1, 0, 0));
    transformGeometry(root(), Matrix4::scale(0.5, 1, 2));
  );
};
```

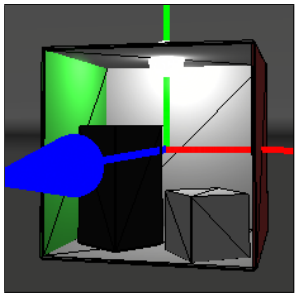

Figure 7: *A rough approximation of the Cornell Box model using seven instances of* `cube.obj`.

You can chose the scale and need not worry about the precise colors and angles. Ensure that the walls have nonzero thickness. I chose 2 cm walls for a 1 m^3 box, which is about the scale of the real 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 compelling 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...

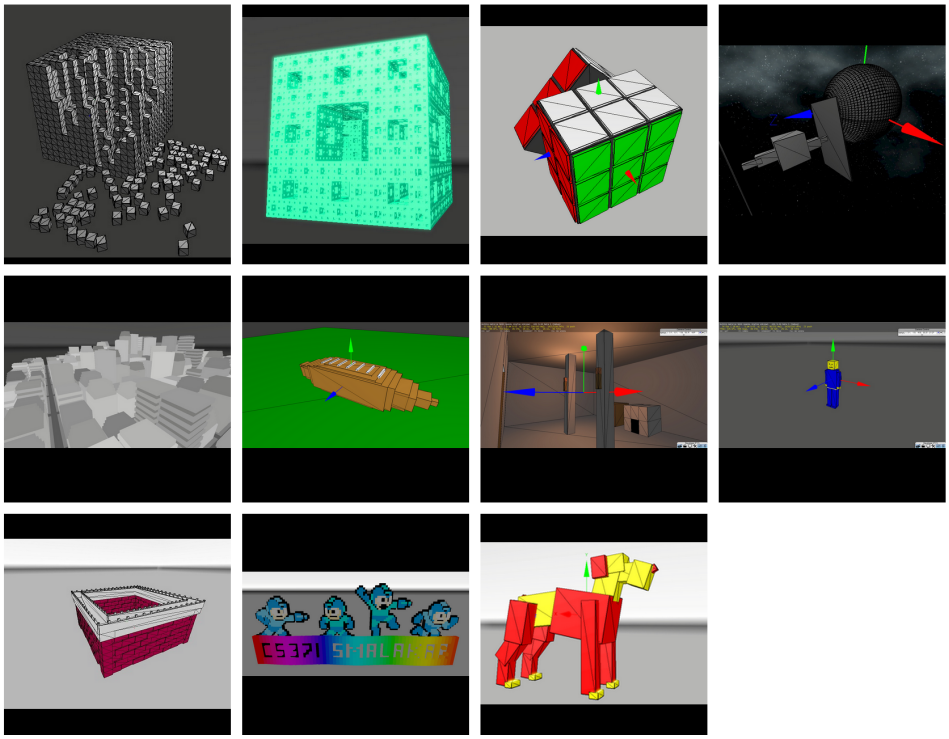

Figure 8: *Custom scenes created from cubes by students in CS371 in 2010.*

4.7 Complete the report and submit

.

Finish off your work. You submit by checking into svn. Whatever is in the repository at the deadline is what I will grade.

Remember to type `svn status` before you leave your computer. Any file marked "M" is modified and not yet committed. Any file marked "?" has never been added. Since you are working in a scratch directory on a local machine, you need to check in any file that you want to see again–anything else will not be available if you use a different computer or if the scratch directory is erased (which happens periodically throughout the semester). You won't complete most projects in a single sitting, so it is a good habit to check in your files whenever you have just completed a big step or when you leave the machine. I'm very paranoid and commit files every time that my build runs successfully–we'll discuss other reasons why this is a good strategy for team projects next week.

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 compelling" 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.

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., 2012. Doxygen 1.8.2 manual. http://www.stack.nl/~dimitri/doxygen/manual.html.

Index