Introduction to Computer Graphics

Computer Science and Engineering
University of California, San Diego

Albert Chern

Last update: October 1, 2021

Contents

1. Introduction

The most creative people are willing to work in the shadow of uncertainty.

Ed Catmull

1.1 What is computer graphics?

Computer graphics are widely seen in the entertainment industry, such as in movies and video games. A lot of 3D animations and visual effects are computer generated. Although the entertainment industry is the mainstream example where computer graphics is best known of, it is not the original motivation in the development of computer graphics. In fact, back in the 1970's, using computer graphics to create movies was a crazy idea pioneered by Ed Catmull.[1]

The early development of computer graphics was driven by improving human-computer interactions. For early computers (from the ENIAC machines in 1945 to PDP-8 machines in the 1960's), it is through punch cards for the input and output of data and instructions. The amount of information one can express through a punch card, light bulbs and digit displays are limited to only several bits in that era. In principle, one can display much more digits of numbers displayed all at once, but humans would not be capable of quickly read and interpret the information.

On the other hand, human brains are very powerful at processing visual data. In fact, 30% of the brain is devoted to visual processing. For us humans, the most efficient form of data to receive information is visual data. Therefore, by representing

[1] Catmull received the 2019 Turing Award for his contributions to 3D computer graphics.

the output information by visual data, we effectively increase the bandwidth in pouring information from the computer output to our mind.

Definition 1.1 **Computer graphics** is the use of computer to synthesize visual (and other sensory) information.

1.1.1 A brief history

Development of computer graphics is an important component of the technological advances of the past 70 years. This brief history gives a glimpse of the evolution of computer graphics and is far from complete.[2]

Cathode ray graphics

Early (50's–60's) examples of visual output device includes the cathode ray tube monitors. They appear on an oscilloscope or on a radar signal display. Though this analog display is primitive from today's standard, the capability of delivering visual data has made breakthroughs in various areas in math and science. For a nontrivial example, the discovery of "soliton solutions" in the Kortweg–de Vries (KdV) differential equation was discovered by *looking* at the solution calculated by a computer, performed by Zabusky and Kruskal in 1965. The discover opens up an entire new math subject of integrable system, which is one of the most significant development in the 20th century mathematics.

Another great example of early computer graphics is a fully computer generated film "Regular Homotopies in the Plane (1974)," which we encourage the reader to take a look. Similar to many math educational videos today, this 1974 film walks through a mathematical concept with the aid of computer generated animation. An interesting detail to notice while watching the film is that the imagery is not shown by pixels (picture elements) like today's televisions and computer monitors. Instead, each curve in the film is generated by the programmed cathode ray directly tracing out the desired geometry.

The rising visual human-computer interactions are not limited to visual output. The *Sketchpad* introduced by Ivan Sutherland in the 1960's is a visual input device. The user can perform complex geometric modeling by controlling the input using a *light pen*.

The rise of arcade video games

A major development in computer graphics took off in the 1970's. This is best seen in the success of arcade video games. The innovation in the arcade video games are the early *graphics processing units* (GPUs), which are computers dedicated for displaying the images on the television screen. It is also the introduction of *rasterized graphics*, which is to visualize the geometry we want to show in terms of the pixels on the TV screen. Video games are also required to process everything in *real time*, that is the calculation time between human input to the video display needs to be shorter than what a human can notice.

The popularity of arcade video games reached its peak in the 1980's, where we see famous games like Pac-Man, many successful games by Nintendo, *etc*.

[2] See also `https://en.wikipedia.org/wiki/History_of_computer_animation`, `https://www.cs.cmu.edu/~ph/nyit/masson/history.htm`

3D computer graphics and space exploration

The University of Utah was a major center for the developments of computer graphics in the 1970's, under the collaboration of Ivan Sutherland and David Evans, and their students. The developments include the foundation of 3D computer graphics, such as the invention of *shading models* (Henri Gouraud, Bui Tuong Phong, Jim Blinn), *Z-buffering*, *texture mapping*, *subdivision*, (Ed Catmull, Jim Clark, *etc.*), *virtual reality*, *flight simulators*, *etc.* A realistic rendering of the 3D model of a human hand by Ed Catmull and Fred Parke is a representing example. Computer generated 3D animated sequences start to appear in popular movies and series in the late 1970's, such as George Lucas' Star Wars (1977) and Carl Sagan's Cosmos (1980) with Ed Catmull, Jim Blinn *etc.* behind the scene. Ed Catmull is also the co-founder of Pixar.

During the time when Jim Blinn worked in NASA Jet Propulsion Laboratory (JPL), he introduced several foundation tools in computer graphics in order to visualize the newest data acquired from space exploration. In 1978, Blinn developed the *bump map* to visualize the new terrain data of Venus. In 1979 the Voyager 1 space probe flied by Jupiter. The physically realistic 3D animation of Voyager and Jupiter was broadcast to the world. The 3D texture reconstruction technique was developed to 3D reconstruct a moon of Jupiter. More sophisticated shading models were developed when the two Voyagers flied by Saturn, Uranus and Neptune in the next decade.

The pipeline for 3D computer graphics was established during that era.

1980's

The personal computer was introduced in the 1980's with graphical user interface. Physically based rendering (by simulating optics via *ray tracing*) emerges in super computer labs. Visually stunning computer animations (*e.g.* Pixar short "Luxo Jr." 1986) show the rapid progress in 3D computer graphics in the 80's.

1990's – Graphics APIs

In the 1990's, there are application programming interface (API) to instruct the GPU, such as OpenGL. The design of the commands in GPU and other units on the graphics card are optimized for the 3D computer graphics pipeline established in the previous decades. Most of these computations in the graphics pipeline are hardcoded and not programmable.

2000's – Programmable shaders

In the 2000's graphics APIs move towards a programmable GPU paradigm. The programmers can code up their own programs in certain stages in the graphics pipeline. Originally, those hard-coded programs run by the GPU are primarily for determining the colors in the pixels. Those programs are thus called *shaders*. Therefore, when the programmers can design their own customized program on the GPUs, we say we have *programmable shaders*.

The customized shaders do not need to perform any shading related computations. One can take advantage of the parallel computing power of the GPU and assign parallel computing task through the shaders. APIs for general purpose GPU programmings appear in this era.

2010's – Real time ray tracing

Most of the 3D graphics pipeline streamlining the programmable shaders is the rasterization-based graphics developed in the 70's. Another paradigm is ray tracing. Rasterization-based graphics is straightforward to be set into a fixed pipeline, with GPU chips designed dedicated to optimize the pipeline for real time performance. However, for a long time ray tracing is regarded as difficult to be made into real time graphics. Ray tracing can produce photorealistic rendering, not realtime, while raster graphics (such as OpenGL) is realtime, but hard to achieve photorealism. In the 2010's, this comparison is no longer the case. There are graphics cards and APIs dedicated to real time ray tracing. That is, we can produce photorealistic imagery in real time. In 2020, multiple game consoles (*e.g.* Playstation 5, Xbox Series X) integrated graphics cards with real time ray tracing. Hence, it is said that we have officially stepped into a new era of computer graphics.

1.1.2 Topics in computer graphics

So far, we have been looking at the development of computer graphics in the aspect of controlling the color in each pixels. But the study in computer graphics is much broader than that.

After all, computer graphics is to synthesize a realistic visual image or animation. To simulate such a virtual digital world, computer graphics is connected to computational physics, differential geometry, optics, *etc.*

- **Rendering** Synthesize photorealistic or non-photorealistic (stylized) image given the geometry, lighting and camera property.
- **Geometry processing** The study of generating, representing, manual editing and automatic optimizing geometric shapes such as curves and surfaces. Subtopics include meshing, computer aided design, geometric optimizations, discretizations of differential geometric theories, *etc.*
- **Image and video processing** The study of synthesizing, modifying, repairing or extracting information from images and videos.
- **Physics simulation** The study of generating computer animation that is based on physics, such as rigid body motions, fluid dynamics, cloth motions, *etc.*
- **Character animation** The study of generating realistic or stylized human or animal animation. The technology includes key frame animations, artificial intelligence, locomotion optimization, motion capture, crowd simulation, *etc.*character animation
- **Visualization** The study of representing data or information using images or videos, *e.g.* data visualization, mathematical visualization, scientific visualization. visualization
- **Fabrication** Producing physical objects that meet real physical constraints. Examples include 3D printing, architectures, woven structures, computer aided knitting, *etc.*fabrication
- **Displays and interaction** Virtual reality, other sensory input and output, *etc.*display and interaction
- **Sound synthesis** Simulation of acoustic phenomena including generating sound effect and their acoustic transfer. sound synthesis

1.2 3D computer graphics: Rasterization v.s. ray casting

The basic task is 3D computer graphics is to represent a 3D scene in a 2D display.

- The 2D display is made of an array of pixels.
- The 3D geometric objects in the scene can have a variety of different digital representations. In any representation, the object is made of many smaller primitive elements, similar to how a 2D screen is composed of many pixels. The most common form of geometry representation is triangle mesh. The surface is made of thousands of triangles stitched together. Each triangle is a primitive element.

Task 1.1 Given 3D geometric objects in a scene, find their occupations in the pixels of the screen.

Once the task is achieved, we will shade the color of those pixels occupied by the geometric object according to the lighting and material property of the object.

There are two techniques that approach Task 1.1, **rasterization** and **ray casting/tracing**, depending on whether we first loop over geometries or we loop over pixels.

1.2.1 Rasterization

The rasterization process projects geometries in the scene to pixels in the scene. Since there may be many geometries occupying a same pixel, as we rasterize in parallel, the output of the rasterization process is a list of fragments piled on pixels before they are sorted per pixel.

Definition 1.2 — Rasterization. Rasterization is the process of computing the mapping from scene geometry to pixels in the form of *fragments*:

```
for each geometric element in the scene do
    for each pixel in the screen do
        if the pixel overlaps the geometric element in the view then
            Generate a fragment associated with the pixel and the geometric element.
        end if
    end for
end for
```

Output: List of fragments.

Mathematically speaking, a *fragment* is a point in the Cartesian product {pixels}×{geometries} of the space {pixels} of all pixels in the screen and the space {geometries} of all geometries in the scene:

$$\text{a fragment} = (\text{a pixel, a piece of geometry}) \in \underbrace{\{\text{pixels}\}}_{\text{screen}} \times \underbrace{\{\text{geometries}\}}_{\text{scene}}. \tag{1.1}$$

Each fragment indicates an incidence (occupancy) between a pixel and a piece of geometry. A list of fragments in this Cartesian space {pixels}×{geometries} therefore represents a mapping (possibly one-to-many and many-to-one), or a table, between the geometries in the scene and the pixels in the screen.

After the rasterization, we will compute a color for each fragment (in parallel) according to the information of the pixel and the geometry properties. Note that there may be many fragments that are associated to a common pixel. At the final step, we decide the final color of the pixel by testing which fragment is the one closest to the screen/camera. This final sorting step is called the *depth test*. If these fragment colors are semi-transparent, then we blend the colors by an *alpha-compositing* technique.

The rasterization process is carried out by fixed, non-programmable hardware within the graphics pipeline. This is because the rasterization method is fixed and there is little motivation for modifying the rasterization algorithm. The special-purpose hardware allows for high efficiency.

1.2.2 Ray casting or ray tracing

The other technique for Task 1.1 is the ray casting. The ray casting process projects the pixels in the screen to the geometries in the scene. Each pixel owns a ray shooting from the eye into the 3D scene through the pixel. The output of the ray casting process produces a ray-geometry intersection for each ray.

Definition 1.3 — Ray casting or ray tracing. Ray casting is the process of computing the mapping from pixels to scene geometry. This is accomplished by tracing rays from the eye, one per pixel, and find the closest object blocking the path of that ray:

```
for each pixel in the screen do
    Generate a ray shooting from the pixel into the 3D scene.
    for each geometry in the scene do
        if the ray of pixel intersects with the geometry, and is closer to the source of the ray compared to previous ray-scene intersection then
            Assign/update the ray-scene intersection to the ray.
        end if
    end for
    Determine the pixel color.
end for
```

The final computation of the pixel color uses the information of the ray-scene intersection. In that sense, a ray-scene intersection is like a fragment in the rasterization paradigm. What is different is that the depth test is done immediately in the inner for loop in the ray tracing. Also, the color of the ray-geometry intersection can be the color of a mirror-reflected ray, or the integral of the colors of a set of scattered rays. By recursively tracing the rays, we effectively simulate the physical process of photons bouncing around in the scene. Therefore, ray tracing is the basis for photorealistic rendering.

In contrast to rasterization which is carried out by a fixed and efficient hardware, the ray tracing routine has to be more flexible and adaptive to the scene. The rasterization can be carried out by scanning over pixels over the screen that are coherently structured, which is good for parallelism; in contrast, the ray tracing has less of such coherency in this scanning/searching task.

Up to 2010, all typical GPUs used rasterization algorithms. Ray tracing renderings were performed “offline” (not real-time or GPU-accelerated). In recent years,

however, hardware acceleration for real-time ray tracing has become standard on new commercial graphics cards.

1.3 Overview of this course

I Rasterization-based Graphics with OpenGL

2. OpenGL Setup

One of the things I really like about programming languages is that it's the perfect excuse to stick your nose into any field. So if you're interested in high energy physics and the structure of the universe, being a programmer is one of the best ways to get in there. It's probably easier than becoming a theoretical physicist.

Bjarne Stroustrup, 2014

In this and the next chapter, we will get familiar with OpenGL. OpenGL and the accompanying shader language GLSL allow us to access the graphics processing unit (GPU) and the graphics card following a rasterization pipeline (Section 1.2.1). OpenGL is one of many graphics application programming interfaces (APIs).[1]

The OpenGL in the 90's and in the early 2000's is called the **Legacy OpenGL** (OpenGL up to 2.0). In Legacy OpenGL, the GPU is not programmable and every function that controls the GPU is used like a black box. We will not use the deprecated Legacy OpenGL. Instead, we study the **Modern OpenGL**, which corresponds to OpenGL 3.0 and later versions. Modern OpenGL allows us to program the GPU. In Modern OpenGL, except for the rasterization step, the pre-rasterization and post-rasterization shading stages of the pipeline are programmable.

[1] `https://en.wikipedia.org/wiki/List_of_3D_graphics_libraries`

Definition 2.1 A **shader** is a program or a piece of code that runs on the GPU.

In particular, the task accomplished by a shader program does not need to relate to the original meaning of shading, which is to determine the color based on lighting and geometry.

2.1 Where is OpenGL? How to load the library?

OpenGL at its core is just a *specification*, or an *agreement* (organized by the Khronos community), of what functions should exist. The actual implementations of these functions are coded by the graphics card manufacturers; *i.e.* the implementations are contained in the driver softwares for their hardwares.[2] In particular, OpenGL is cross-platform, but it is not open source (despite the word "open" in its name) since a graphics card vendor is likely not sharing the secrets for why their implementation performs faster.

Task 2.1 To develop graphics applications with OpenGL, we need to
1. use the OpenGL library that allows us to instruct the graphics card;
2. use a utility library that help us open a window and detect our keyboard and mouse activities.[a]

[a] Note that OpenGL only controls parts of the graphics card, and does not help us with the tasks depending on the operating system such as opening a window.

Tip If you are not familiar with the concept of including library headers and linking library binaries, see Appendix A.

2.1.1 Overview

We give a quick overview what libraries are expected to be included an linked. In the next subsections, we explain the existence of the libraries in each platform. Finally we summarize the installation by building a "Hello Window" application.

The GL Part

The first task of Task 2.1 is to **load OpenGL**. We will *effectively* include the following headers (and load the associated library binaries):

- `<GL/gl.h>` – The legacy OpenGL features.
- `<GL/glu.h>` – Additional high-level functions such as `gluLookAt`, `gluBeginPolygon`, `gluEndPolygon`, *etc*. They are utilities for Legacy OpenGL. We will not rely on these functions in Modern OpenGL covered by this lecture note.
- `<GL/glext.h>`, `<GL/wglext.h>`, `<GL/glcorearb.h>`, `<GL/glxext.h>`, `<GL/gl3.h>` and possibly other headers – They contain additional OpenGL specifications enabling Modern OpenGL features. How Modern OpenGL features are imple-

[2] For MacOS, all graphics drivers are part of the Mac system, instead of an independent software maintained by the graphics card manufacturers. Therefore it is up to Apple whether they want to support OpenGL. Currently (2021) Mac machines (including the ones using the M1 chip) support OpenGL 4.1. Some of these OpenGL 4.1 implementations are through Metal, Mac's graphics API.

mented depends on the graphics card driver. This makes the inclusion and linking of the libraries vary over different platforms and hardwares.

As we can see, it seems challenging to write an appropriate cross-platform inclusion of the Modern OpenGL. Fortunately,

- On MacOS, Apple implements their own version of OpenGL that does not rely on the graphics card party. With such a full control, MacOS decides that to include the Modern OpenGL functionalities, we just include the single header `<OpenGL/gl3.h>`.
- For Windows and Linux, we can use the open-source library **OpenGL Extension Wrangler** (GLEW) that helps us load the OpenGL extensions. To use the Modern OpenGL functionality, we just include the single header `<GL/glew.h>` (we will need to install GLEW).

The GLUT Part

The second task of Task 2.1 means that we want to load **GLUT** (**OpenGL Utility Toolkit**). GLUT is a classic library cross-platform API that handles the window and detects the keyboard/mouse events. (Another frequently seen utility library similar to GLUT is GLFW. But we will stick with GLUT.) Note that GLUT is not maintained and updated for many years, but it is still usable for the basic tasks.[3][4] An open-source replica **FreeGLUT** is available, which is maintained and updated. We expect to include

- `<GL/glut.h>`

and link its associated binary.

2.1.2 Windows Platform

The headers `<GL/gl.h>` *etc.* and the associated binary `opengl32.lib` are part of **Windows SDK** (Windows software development kit). You can install Windows SDK by selecting it in the optional components of the Visual Studio Installer. While the essential OpenGL are there, we need to manually install GLEW and GLUT (or FreeGlut). You can download their pre-built libraries from their websites (*e.g.* the freeglut MSVC[5] releases).

We will be including and linking GLEW and GLUT relatively. Assume that we are running on a 32bit system.

Create an `include` and `lib` folder in our project. In `include`, create a `GL` folder. Put all the GLEW and GLUT header (`.h`) files in our `GL`. Put all the static library (`.lib`) files to the `lib` folder. Put all the dynamic library (`.dll`) files in the same directory where our executable (`.exe`) is expected to appear. See Figure 2.1 (a).

In Visual Studio, set the project (solution) property with

- Configurations: `All Configurations`; Platform: `Win32`.
- C/C++→General→Additional Include Directories: `$(SolutionDir)\include\`.
- Linker→General→Additional Library Directories: `$(SolutionDir)\lib\`

[3] The only annoying part is that GLUT does not adapt with the high DPI screens like retina display appeared in the 2010's. It does not really bother us most of the time.

[4] Both GLUT and the similar toolkit GLFW are suitable for small to medium OpenGL applications and learning OpenGL. For developing larger applications, it is better to use the platform-native toolkit, like the Win32 API for Windows, X Window System (X11) for Linux, and Cocoa for MacOS.

[5] Compiled by Microsoft Visual C++.

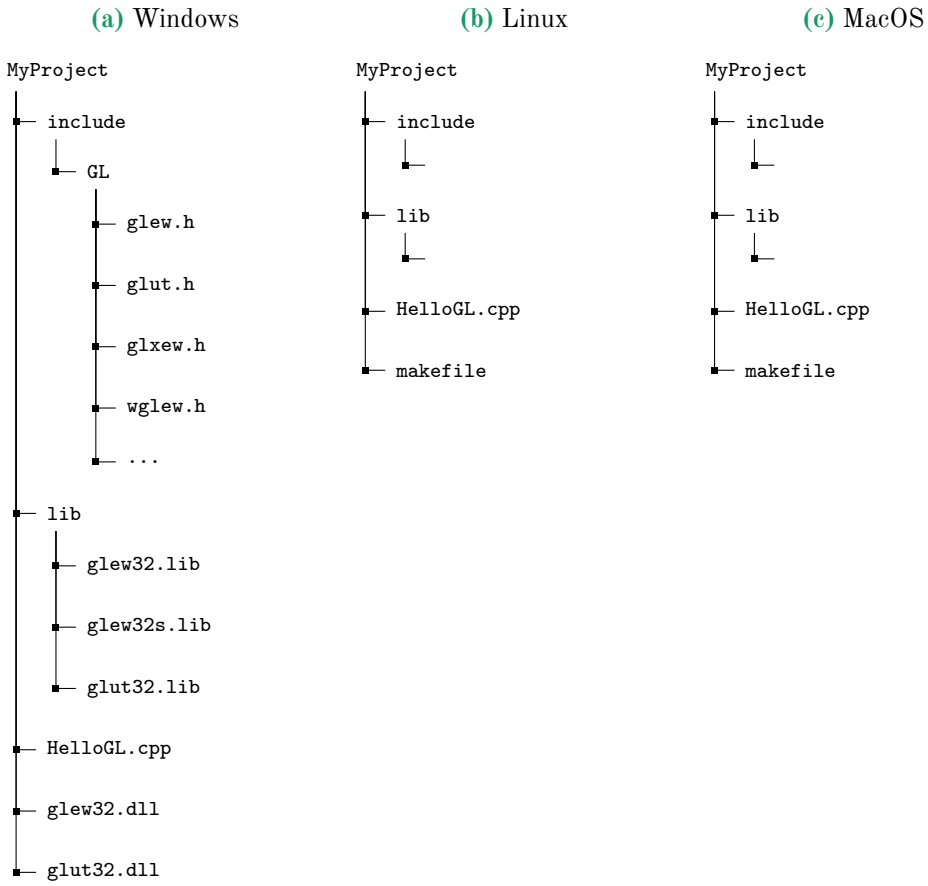


Figure 2.1 Recommended directory structures for an OpenGL project on various platforms.

- Linker→Input→Additional Dependencies: `opengl32.lib`,[6] `glew32s.lib`,[7] `glut32.lib`.

In our C++ file that uses OpenGL and GLUT, say `HelloGL.cpp` in Figure 2.1 (a), the header will be

```
#include <GL/glew.h>
#include <GL/glut.h>
```

2.1.3 Linux Platform

On Linux, we need the headers `<GL/gl.h>`, *etc.*, and we need to link with `libGL.so` (which is a symlink to the library in the graphics driver). The headers and the library files can be installed with the Mesa Package. Mesa is an open-source implementation of the OpenGL specification compatible with a large variety of hardwares.

On Ubuntu, we use the `apt-get` package manager. Other systems may use a different package manager. Open a terminal and type

```
sudo apt-get update
sudo apt-get install build-essential # make and g++, if you haven't already
```

[6] Recall that `opengl32.lib` is given by Windows SDK and is already in the search path.
[7] The "`s`" in `glew32s.lib` indicates that we are choosing the static linking.

```
sudo apt-get install mesa-common-dev # OpenGL
sudo apt-get install libglew-dev     # GLEW
sudo apt-get install freeglut3-dev   # GLUT
```

In principle, we should see those header files in `/usr/local/include/GL/` and the binaries in `/usr/local/bin/`.

We will include and link the libraries from these system directories. But we will still create empty local directories named "`include`" and "`lib`" as placeholders for more libraries later (Figure 2.1).

We will build the project with makefile. Here is an example of the makefile showing the library dependency:

Code 2.1 `makefile` on Linux

```
# Macro
CC = g++
CFLAGS = -g
INCFLAGS = -I./include/ -I/usr/local/include/
LDFLAGS = -L./lib/ -L/usr/local/bin/ -lGL -lGLU -lGLEW -lglut
RM = /bin/rm -f

all: HelloGL
# Linking
HelloGL : HelloGL.o
    $(CC) -o HelloGL $(LDFLAGS) HelloGL.o
# Compilation
HelloGL.o : HelloGL.cpp
    $(CC) $(CFLAGS) $(INCFLAGS) -c HelloGL.cpp

clean :
    $(RM) *.o HelloGL
```

In our C++ file that uses OpenGL and GLUT, the header will be

```
#include <GL/glew.h>
#include <GL/glut.h>
```

2.1.4 MacOS Platform

On MacOS, we include and link Mac's native OpenGL and GLUT frameworks. These frameworks come with **macOS SDK** (software development kit). MacOS SDK is contained in the **command line tools**. Command line tools also consist essential tools such as `g++`, `make`. So, if you have not done so already, install the command line tools.

Install the command line tools

First, check whether the command line tools have been installed. Just check whether the directory exists:

```
/Library/Developer/CommandLineTools/
```

If it exists, then you have the command line tools. If not, take one of the three options:

- Install the package manager **Homebrew**.[8] (To check whether homebrew has

[8] Even if you already have the command line tools, we still recommend that we install Homebrew for other library installations.

been installed, open a terminal and type "`brew`". If we see "`Command not found: brew`" then homebrew has not been installed.) Go to homebrew's website and copy the installation command to your terminal. If we haven't installed the command line tools, during the homebrew installation we will see a message "The Xcode command line tools will be installed." Press return to continue.

- Just type

```
xcode-select --install
```

in a terminal. A pop-up dialog will show up; click "install." This will install the command line tools without installing the full-blown Xcode App.

- Install the full Xcode App from App Store. It should give us the command line tools. (In an older Xcode, we would have to further go to the Preferences of Xcode and download the command line tools component.)

OpenGL on MacOS

Now, OpenGL and GLUT are available in the directories

```
/Library/Developer/CommandLineTools/SDKs/ \
    MacOSX.sdk/System/Library/Frameworks/OpenGL.framework
/Library/Developer/CommandLineTools/SDKs/ \
    MacOSX.sdk/System/Library/Frameworks/GLUT.framework
```

These `.framework` directories contain the headers and the binaries, and they are already in system's search path.

Although there are no further dependencies at this moment, we will still create empty local directories named "`include`" and "`lib`" in our local project folder as placeholders for more libraries later (Figure 2.1).

We will build the project with makefile. Here is an example of makefile showing the library dependency:

Code 2.2 `makefile` on MacOS

```
# Macro
CC = g++
CFLAGS = -g -Wno-deprecated-register -Wno-deprecated-declarations
INCFLAGS = -I./include/
LDFLAGS = -L./lib/ -framework OpenGL -framework GLUT
RM = /bin/rm -f

all: HelloGL
# Linking
HelloGL : HelloGL.o
    $(CC) -o HelloGL $(LDFLAGS) HelloGL.o
# Compilation
HelloGL.o : HelloGL.cpp
    $(CC) $(CFLAGS) $(INCFLAGS) -c HelloGL.cpp

clean :
    $(RM) *.o HelloGL
```

Currently, Apple view OpenGL as deprecated (not maintained or updated). So the compilation produces many warnings when the OpenGL functions are called. The `CFLAGS`

```
-Wno-deprecated-register
-Wno-deprecated-declarations
```

are there to silence the warnings.

In our C++ file that uses OpenGL and GLUT, the header will be

```
#define GL_DO_NOT_WARN_IF_MULTI_GL_VERSION_HEADERS_INCLUDED
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
```

The header file `gl3.h` indicates that we are working with Modern OpenGL. However, there will be many warning coming from the fact that `glut.h` will include the Legacy OpenGL (`gl.h`, `glu.h`) as well. The definition (as a preprocessor flag)

```
#define GL_DO_NOT_WARN_IF_MULTI_GL_VERSION_HEADERS_INCLUDED
```

is there to silence these warnings.

2.1.5 Summary

Summary 2.1 The following table summarizes the setup on each platforms:

	Windows	Linux	MacOS
OpenGL	Comes with Windows SDK.	Comes with the Mesa graphics package.	Comes with MacOS SDK in command line tools.
GLEW, GLUT	Headers copied to `$(Proj)\include\GL\`, `.lib` files copied to `$(Proj)\lib\` and `.dll` files copied to `$(Proj)` next to our executable.	Installed with a package manager such as `apt-get`. The headers are in `/usr/local/include/` and the binaries are in `/usr/local/bin/`	GLUT comes with the SDK. We don't need GLEW.
Additional include directories (`-I`)	`$(Proj)\include\`	`/usr/local/include/`	
Additional library directories (`-L`)	`$(Proj)\lib\`	`/usr/local/bin/`	
`#include`	`<GL/glew.h>` `<GL/glut.h>`	`<GL/glew.h>` `<GL/glut.h>`	`<OpenGL/gl3.h>` `<GLUT/glut.h>`
Link	`opengl32.lib`, `glew32s.lib`, `glut32.lib`	`GL`, `GLU`, `GLEW`, `glut`	`OpenGL` and `GLUT` as frameworks.

2.2 Hello Window

The first exercise is to create a blank window.

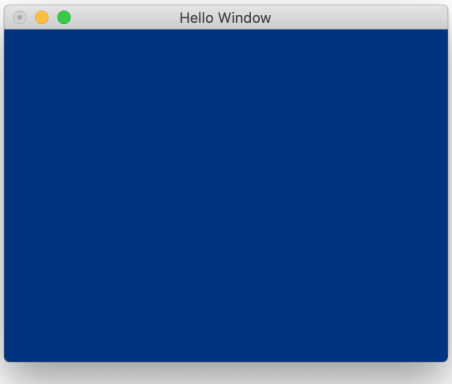


Figure 2.2 Result of Code 2.3.

Code 2.3 `HelloGL.cpp`

```
#ifdef __APPLE__
#define GL_DO_NOT_WARN_IF_MULTI_GL_VERSION_HEADERS_INCLUDED
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
#else
#include <GL/glew.h>
#include <GL/glut.h>
#endif

#include <iostream>

static const int width = 400;
static const int height = 300;

void display(void){
  glClear(GL_COLOR_BUFFER_BIT);
  glutSwapBuffers();
  glFlush();
}

void keyboard(unsigned char key, int x, int y){
  switch(key){
    case 27: // Escape to quit
      exit(0);
      break;
  }
}

int main(int argc, char** argv)
{
/* Begin Create Window */
  glutInit(&argc, argv);

#ifdef __APPLE__
```

```
35    glutInitDisplayMode( GLUT_3_2_CORE_PROFILE | GLUT_DOUBLE | GLUT_RGB |
        GLUT_DEPTH);
36  #else
37    glutInitContextVersion(3,1);
38    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
39  #endif
40
41    glutInitWindowSize(width, height);
42    glutCreateWindow("Hello Window");
43
44  #ifndef __APPLE__
45    glewExperimental = GL_TRUE;
46    GLenum err = glewInit() ;
47    if (GLEW_OK != err) {
48        std::cerr << "Error: " << glewGetErrorString(err) << std::endl;
49    }
50  #endif
51
52    std::cout << "OpenGL Version: " << glGetString(GL_VERSION) << std::endl;
53  /* End Create Window */
54
55    std::cout << "Press ESC to quit." << std::endl;
56    glClearColor (0.0, 0.2, 0.5, 1.0); // background color
57    glViewport(0,0,width,height);
58
59    glutDisplayFunc(display);
60    glutKeyboardFunc(keyboard);
61
62    glutMainLoop();
63    return 0;
64  }
```

In the block

```
/* Begin Create Window */
...
/* End Create Window */
```

GLUT creates a window and the final buffers at the display end (with options "`GLUT_DOUBLE`" *etc.*).

Outside this block, GLUT manage the keyboard events by callbacks. Internally, there is a main infinite loop (`glutMainLoop()`). Before the main loop, which is the initialization stage, we set up which function should be called when the keyboard is pressed and when the display routine is triggered.

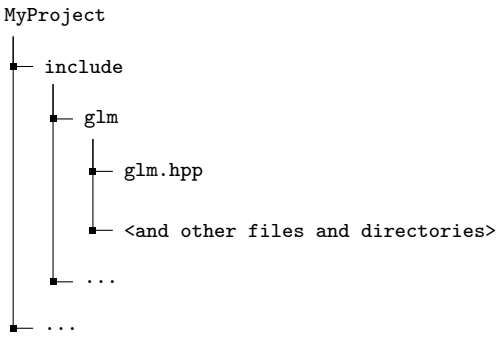
2.3 GLM and FreeImage

We will install two more libraries, GLM and Freeimage.

GLM is the OpenGL math library. It provides types like `glm::vec2` (2D vectors), `glm::mat4` (4×4 matrices) *etc.*, so that you can manipulate graphics related mathematics at a more intuitive level. In fact, these math types are exactly the available types in the shader language GLSL for the GPU programming. GLM is the C++ library so that your C++ code looks similar to a GLSL code.

GLM is a header-only library. There is no binary to link. You can put the GLM

folder in the local include directory:



You can also install GLM into the system. For example, on Ubuntu `sudo apt-get install libglm-dev`, and on Mac `brew install glm`. Make sure that the `glm` folder is in the search path for compilation. To include the library in our code

```
#include <glm/glm.hpp>
```

FreeImage is an Open Source library that let us export image files like `.jpg`, `.png`. The main purpose of having the library is to produce screenshots directly from the pixel data.

On Windows, put `FreeImage.dll` next to other `.dll` files next to the executable. Put `FreeImage.lib` in the `MyProject\lib\` folder. Put `FreeImage.h` in the `MyProject\include` folder. Add `FreeImage.lib` to the additional dependencies.

On Mac and Linux, put `FreeImage.h` in the `./include`, and put `libFreeImage.a` in `./lib`. Make sure to have `-I ./include` for the compiler, and `-L ./lib` for the linker. (You can also install FreeImage with package manager like `sudo apt-get install libfreeimage-dev` or `brew install freeimage`.) Finally, make sure to have `-lfreeimage` for the linker.

In our code:

```
#include <FreeImage.h>
```

3. OpenGL: a Graphics Factory

A display connected to a digital computer gives us a chance to gain familiarity with concepts not realizable in the physical world. It is a looking glass into a mathematical wonderland.

Ivan Sutherland, 1965

In this chapter, we dive into the OpenGL API and GLSL. The OpenGL is a **state machine**. It simulates a graphics "factory."

3.1 A tour of the graphics factory

The raw ingredients that go into the factory are the data describing 3D geometries. The data is in the form of a spreadsheet of numbers. The final product that comes out of the factory is an array of pixel colors to be shown on the display. Between the raw ingredients and the final product is the so-called **rendering pipeline**.

We, who will be the commander of the factory, can select a mode called "Triangles" and press a button labeled "Draw Elements" (`glDrawElements( GL_TRIANGLES, ... );`). Then, the factory will perform one cycle of the pipeline routine, grabbing a spreadsheet of geometric data and producing the output pixel colors. There are only 12 modes to select from, and `GL_TRIANGLES` is only one most graphics production will use. The majority of 3D geometric surfaces are converted to triangle meshes for OpenGL-like rendering.

The overall pipeline routine triggered by one "Draw Elements" command consists of a fixed sequence of procedures. From the input geometric data to the final pixel colors, the procedures are organized in the following stages:

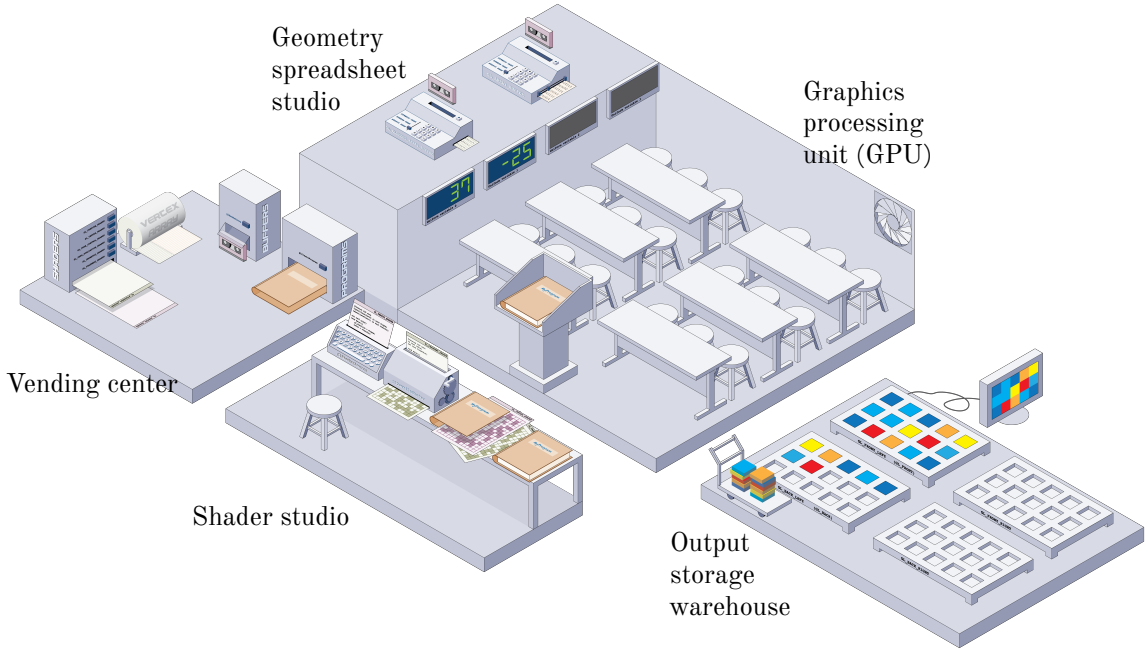


Figure 3.1 The graphics hardware is illustrated as a graphics factory. OpenGL commands are operations that manipulate the objects and the machines in the factory.

(i) **Pre-rasterization stage.** According to the input data, represents the 3D geometries in a canonical coordinate system (called **normalized device coordinate**).
(ii) **Rasterization.** Assembles the 3D geometry elements (triangles), discards the ones outside the "visible box" in the canonical coordinate, and rasterizes the remaining elements into fragments (Section 1.2.1).
(iii) **Post-rasterization stage.** Paints each fragment with a color, and export the fragment colors into the final output buffer for storage and delivery (display).

3.1.1 The GPU in the factory

Each of the above stages is a computation task performed on the GPU. In the analogy of our graphics factory, GPU is a unit at the heart of the factory, illustrated in Figure 3.2. It consists of many workers who will execute the *same* instruction in parallel. The instructions are written in a **program**, which can be written by ourselves (visualized as the book on the podium in Figure 3.2). These instructions are usually arithmetics, *e.g.* adding numbers. The only nuance is that the workers are fed with different input data, so the output data can be different even when the workers execute the exact same instruction. That is, the type of task the GPU can accomplish is categorized as **single instruction, multiple data** (**SIMD**).

In the GPU in our graphics factory, we also see a few big numerical displays showing numbers that all workers can look up. These numbers are called **uniform variables**. If a piece of data that needs to be processed by the workers are the same for all workers, then we just show the data as uniform variables, instead of duplicating them and distributing them to each workers' desk.

What are the workers in the GPU doing for each of the (pre/post-)rasterization

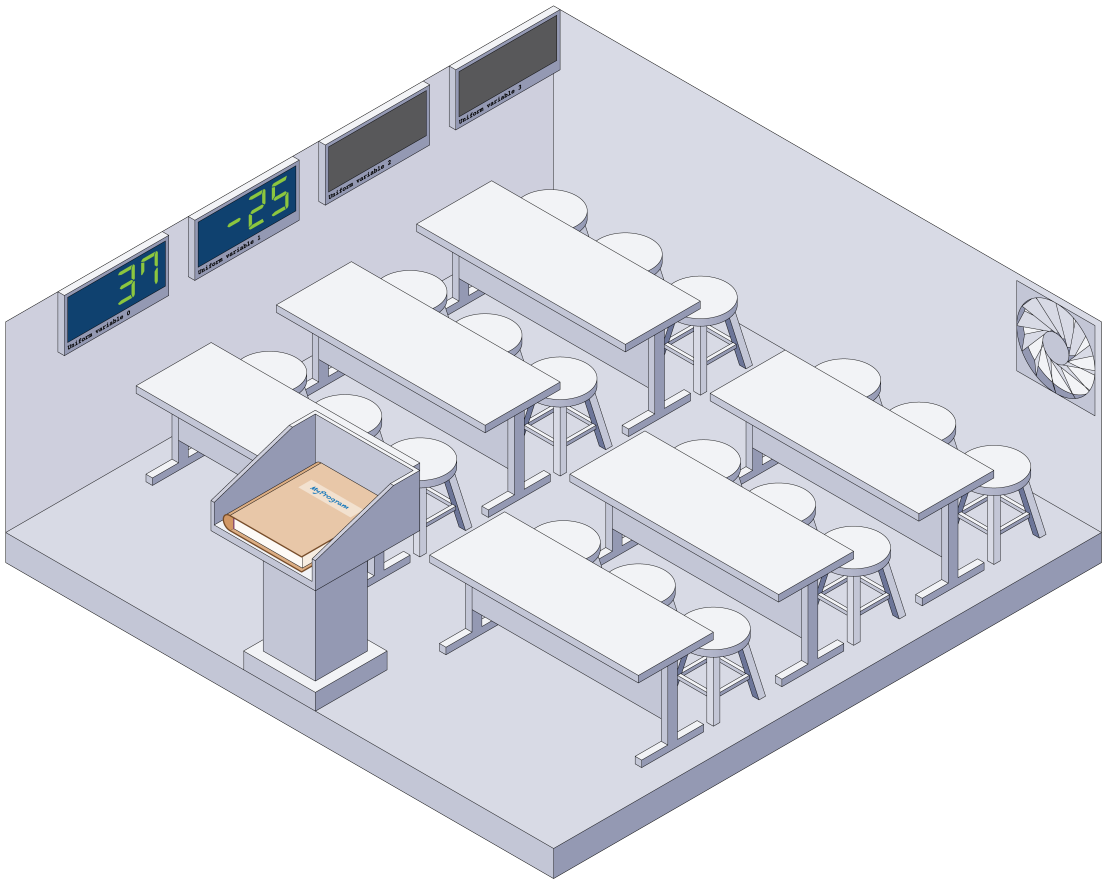


Figure 3.2 The graphics processing unit (GPU) in the graphics factory.

stages?

In the (i) pre-rasterization stage, the workers compute the coordinate values[1] for each vertex of the triangle mesh. To do so, they follow a set of instruction provided by us. This instruction set is called the **vertex shader**, which is a program executed per vertex. The source code for the vertex shader is written in a C-like language called **GLSL**. Since we get to write our own vertex shader, we can ask the GPU to perform much more than just computing the vertex coordinates.

The (ii) rasterization stage is usually not programmable. It consists of fixed routine that determines the visibility of each triangle from each pixel. The output of this stage is a set of fragments. Each fragment is endowed with customized data/information prescribed in the vertex shader.

In the (iii) post-rasterization stage, the workers compute the fragment color according to the data on each fragment. The instruction for this per-fragment computation is provided by us in the form of the **fragment shader**. It is also written in GLSL.

This is why OpenGL is a **state machine**. It is always the same function

```
glDrawElements(GL_TRIANGLES,...)
```

[1] under the normalized device coordinate system the rasterizer is gauged into.

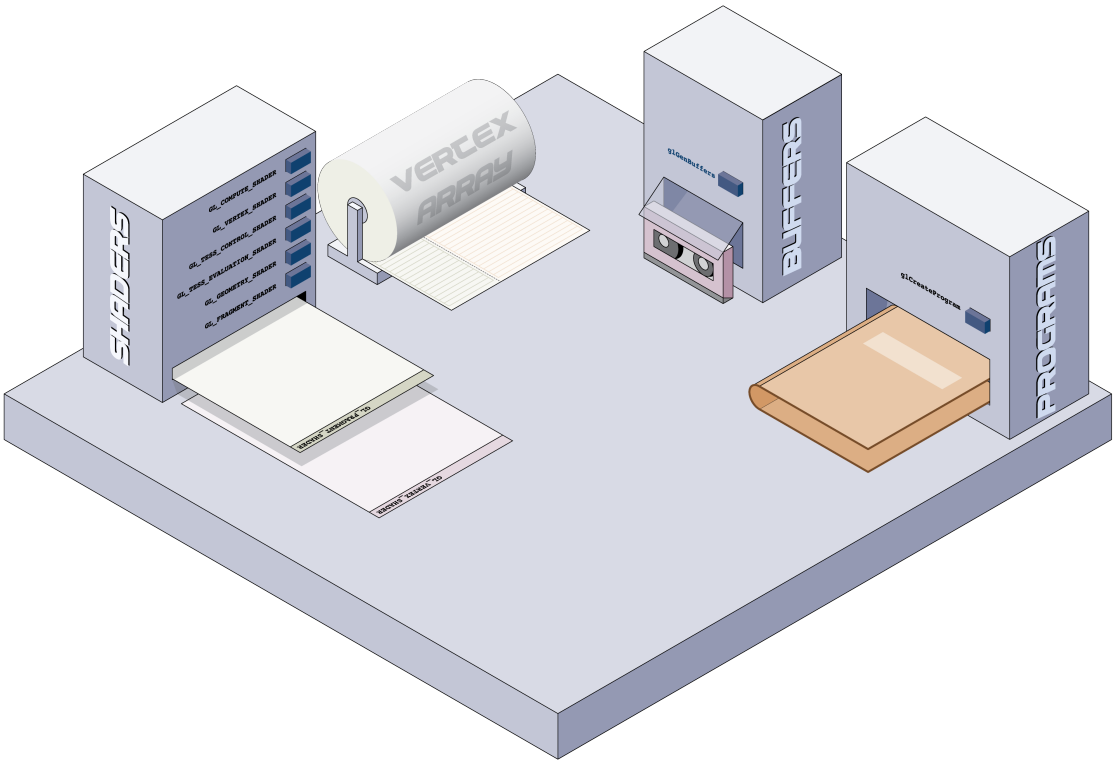


Figure 3.3 The "vending machines" from which we can create the following objects: buffer objects (the casette tapes), vertex array objects (the spreadsheets), shader objects (sheets of paper from the shader vending machine), and programs (the book cover).

for producing any image. But this function alone does not determine the final picture. What determines the actual picture on the image is the state of which spreadsheet of input data we bind and which shader program we use.

3.1.2 The objects we will work with

The setup of the state machine include two main components: setting up the geometry data and setting up the shader program. To manipulate them, we primarily operate with the following four OpenGL objects:

- **Buffer object**.
- **Vertex array object**.
- **Shader object**.
- **Program object**.

We will often see the word **buffer** in the computer graphics pipeline:

> **Definition 3.1** A **buffer** is an allocated part in the memory[a] for storing data for later access.
>
> [a] such as the video RAM on the graphics card.

Think of a buffer object as a "cassette tape" (Figure 3.3). Each buffer object stores a linear sequence of data. In the "vending center" of our graphics factory stands a

buffer vending machine with a button labeled `glGenBuffers`, supplying empty cassette tapes.

The buffer objects can store data, but they alone are difficult to represent geometry. A geometry, such as a triangle mesh, are represented with a *formatted* data. Rather than a long array of numbers stored in a buffer, we need to layout the numbers in tables. One of the tables is organized by that each row corresponds to a vertex and each column corresponds to an attribute (for example the vertex position is an attribute). The other table describes the connectivity among the vertices to characterize the triangle mesh. These formatted data are often called **geometry spreadsheets** in many graphics applications. In OpenGL, the concept of geometry spreadsheet is called **vertex array**.

Definition 3.2 In OpenGL, a geometry spreadsheet is called a **vertex array object**.

In reality, the geometry spreadsheet (vertex array object) is merely a set of pointers that redirect the spreadsheet reader to the correct locations of the correct buffers.

In our graphics factory illustration (Figure 3.3), a new vertex array object is visualized as an empty sheet of paper to be printed with two tables of numbers (the vertex attributes and the triangle connectivity).

Finally, we have the shader object and the program object. A shader object is classified into different types, such as the vertex shader type and the fragment shader type. A shader object is where we write GLSL source code. The shader objects will be compiled and then linked into a single program. The program object is this final target of the linking.

In our illustration (Figure 3.3), shader objects are the official pieces of paper on which we write the source code. These papers are to be translated into a machine code and print-pressed into a book (program object). A new program object is illustrated as an empty book cover to be filled with linked machine code.

3.1.3 The geometry spreadsheet studio

Our next stop of the tour is the "geometry spreadsheet studio" (Figure 3.1).

As mentioned earlier, a geometry spreadsheet has the information that can characterize the shape and other properties of a geometry. The geometry spreadsheet consists of a "vertex attribute table" and a "connectivity table." For the vertex attribute table, the rows of the table correspond to the vertices of a geometry, and the columns of the table are the vertex attributes. For example, some of the vertex attributes correspond to the vertex position. Some vertex attributes may describe the vertex color.

The connectivity table describes the connectivity of the vertices when they are assembled into a triangle mesh. This connectivity table has many rows, but only three columns; the entries of the table are all unsigned integers. Each row of the table correspond to a triangle, and the three integers of that row indicates which the three vertices this triangle is attached to.

A geometry spreadsheet containing the "vertex attribute table" and the "triangle connectivity table" is all we need for feeding into the rendering pipeline.

In our factory, we have a geometry spreadsheet studio. It is an office where we can work with the geometry spreadsheet.

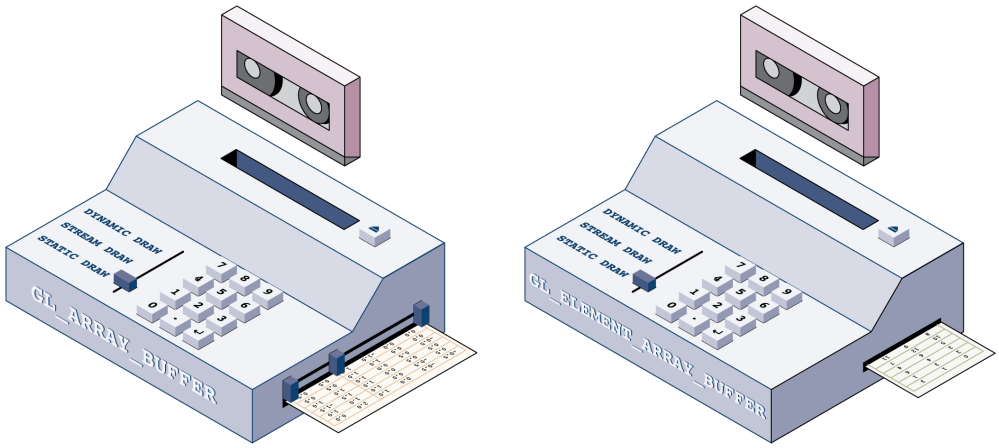


Figure 3.4 The two buffer object readers and writers, one for assigning vertex attribute (`GL_ARRAY_BUFFER`, left) and one for assigning triangle connectivity (`GL_ELEMENT_ARRAY_BUFFER`, right).

In our spreadsheet studio, there are two machines for accessing cassette tapes (Figure 3.4). One is labeled

```
GL_ARRAY_BUFFER  // for vertex attributes
```

and the other is labeled

```
GL_ELEMENT_ARRAY_BUFFER // for connectivity
```

Each of them has only *one* slot for inserting a cassette tape. Both of them let us read/write data from/into the cassette tape.

While a tape is inserted in the `GL_ARRAY_BUFFER` machine, we can configure a portion of the buffer as a customized vertex attribute. By doing so, we fill the vertex array object (geometry spreadsheet).

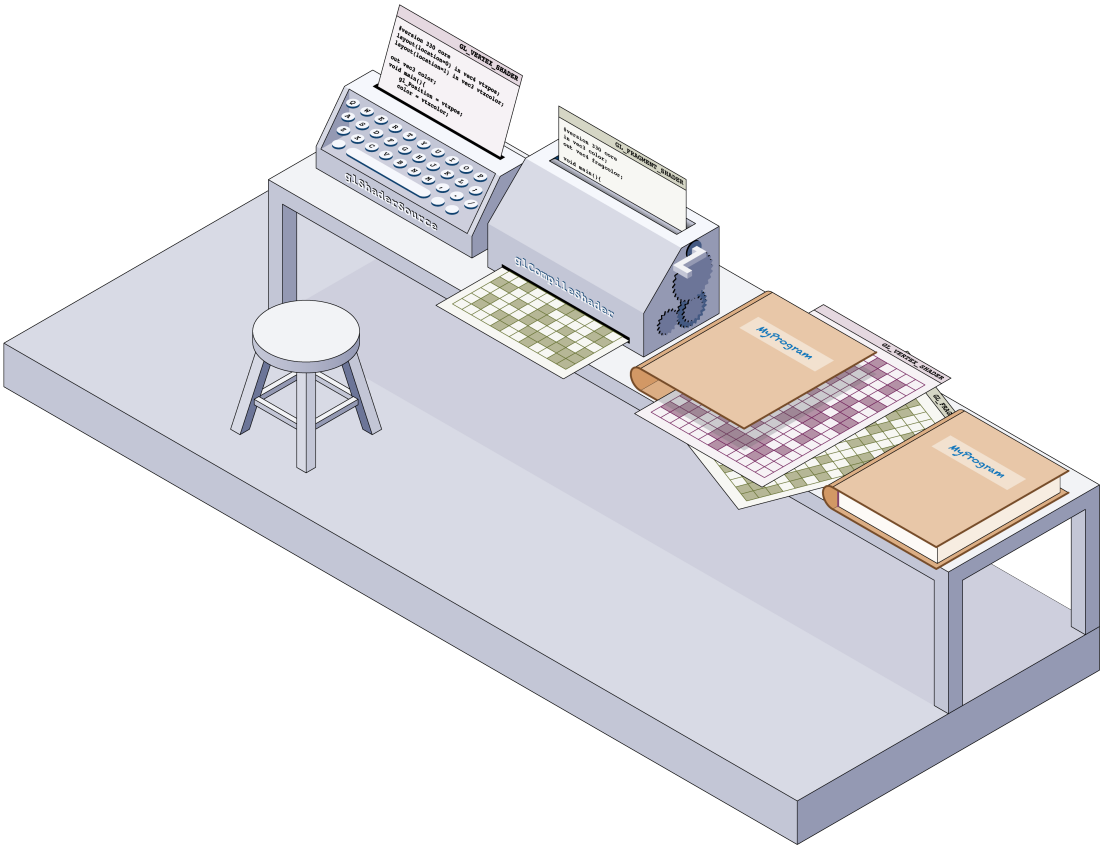
3.1.4 The shader studio

In our graphics factory, there is a shader studio. In this studio, we will work with writing and compiling our shader program. In this office, there is a desk. On top of the desk are a typewriter and a translator.

The typewriter is labeled `glShaderSource`. This typewriter will be the equipment for typing the source code (GLSL). The translator, labeled `glCompileShader`, takes a piece of paper with GLSL on it, and translates the GLSL code into a binary code on the same piece of paper. Several pieces of binary code are bound together and printed into a book through the process of `glLinkProgram`.

The source code needs to be typed on special pieces of paper. These official pieces of paper are the shader objects, which we can obtain from the vending center (Section 3.1.2). There are many selections on the vending machine, but we will only take the `GL_VERTEX_SHADER` and the `GL_FRAGMENT_SHADER`. Similarly, there is another vending machine right next to it, labeled `glCreateProgram`. This vending machine supplies blank books (program objects).

After we print the compiled shader code onto one of these books, the book becomes the shader program that can be put on the podium in the GPU.



3.1.5 The framebuffers

The final stop of our tour is at the output end of the rendering pipeline. Like many factories in real life, the output end is a warehouse for storage and delivery the products (Figure 3.5). The product of the rendering pipeline is in the form of fragment colors. These outputs are stored in a buffer whose size is the image we want to display. This buffer is called a **framebuffer**.

Double buffering

There are several framebuffers that are working at the same time. For example, the technique of **double buffering** requires two buffers, one is the called the **front buffer** (`GL_FRONT`) and the other is called the **back buffer** (`GL_BACK`). The front buffer is the frontstage. It delivers its pixel colors to the display. In contrast, the back buffer is the backstage. As the pipeline is producing the fragment output, the colors are kept updated in the backstage. Only when we call `glutSwapBuffers()` that the front buffer and the back buffer are swapped.

Without double buffering, we would see on the display that each geometric objects are drawn one by one as they are rendered, and therefore makes the display look unpleasantly flashy.

Depth buffer

When a fragment of a particular pixel comes out of the rendering pipeline, it is either replacing the corresponding pixel in the designated frame buffer, or it discarded. The latter happens when there has already been a fragment sitting in that pixel that is

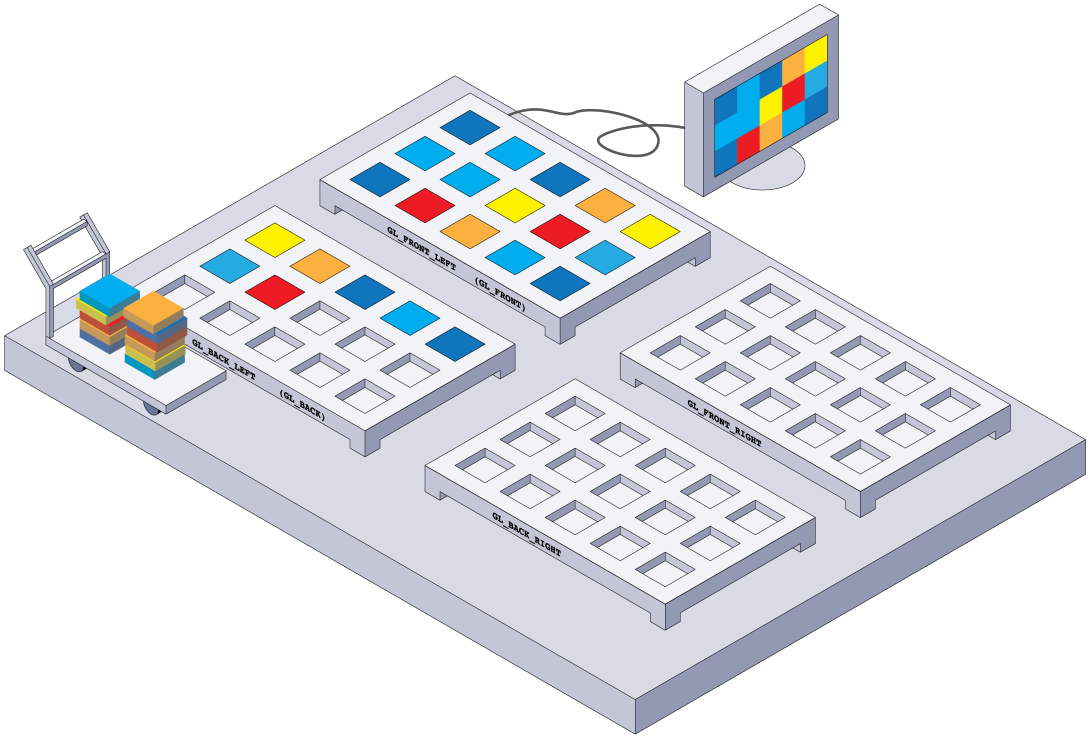


Figure 3.5 The output storage and delivery warehouse consisting of framebuffers.

closer to the camera. In order to compare this *depth* value, we use a **depth buffer** to store the current depth value of the pixel in the frame buffer.

If a fragment has a depth value smaller than the current depth value in the depth buffer, than it should replace the pixel in the frame, and update the value in the depth buffer. Otherwise, the fragment is hidden by the current frame pixel, so we can discard the fragment.

If the fragment has transparency, then a single depth buffer is not enough to compute the correct transparent-color blending. In that case, a larger amount of memory per pixel is required.

3.1.6 Finishing the tour

To operate the graphics factory:

Task 3.1 Our task is to
1. provide a geometry spreadsheet describing our geometry,
2. write a vertex shader, and
3. write a fragment shader,

so that when we call `glDrawElements(GL_TRIANGLES,...)` the desired picture shows up on our screen.

3.2 Rasterization as a projection

In the next section, we will draw a square. We will create a square, made of two triangles, in the 3D space. And then we will let the rasterizer project it onto the image plane. In order to do so, it is crucial that we understand what kind of mapping the rasterization really does.

Definition 3.3 — Viewing box. The **viewing box** in the **normalized device coordinate system** is the 3D window

$$\mathbb{V}_{3D} := [-1,1] \times [-1,1] \times [-1,1] \subset \mathbb{R}^3. \tag{3.1}$$

Definition 3.4 — Normalized screen coordinate. The **viewport** in the **normalized screen coordinate** is the planar square region

$$\mathbb{V}_{2D} := [-1,1] \times [-1,1] \subset \mathbb{R}^2. \tag{3.2}$$

What the rasterizer in OpenGL does is to project each point in the viewing box in the normalized device coordinate to the window in the normalized screen coordinate. This projection is simply removing the z coordinate.

Definition 3.5 — Rasterizer's orthographic projection. The rasterizer's projection is the map

$$\begin{aligned} P_{\text{rast}} \colon \mathbb{V}_{3D} &\to \mathbb{V}_{2D} \\ (x,y,z) &\mapsto (x,y). \end{aligned} \tag{3.3}$$

When the depth buffer is enabled, we use the z coordinate to compare the relative closeness of a point to the screen. For $(x,y,z_1),(x,y,z_2) \in \mathbb{V}_{3D}$ that are mapped to the same point in $\mathbb{V}_{2D}$ by P_{rast}, we say that (x,y,z_1) is *closer to the screen (eye, camera) than* (x,y,z_2) if $z_1 < z_2$.

Note that (3.3) is the only kind of the 3D-to-2D projection the rasterization will ever do. In particular, it does not give us perspective foreshortening (objects further from the camera look smaller). In a later chapter, we will learn how to place the 3D geometry in $\mathbb{V}_{3D}$ in such a way that the view looks perspective upon rasterization.

3.3 Hello Square

We start with Code 2.3. The main function is similar to that of Code 2.3:

Code 3.1 `HelloSquare.cpp main()`

```
1 int main(int argc, char** argv)
2 {
3   // BEGIN CREATE WINDOW
4   glutInit(&argc, argv);
5
6 #ifdef __APPLE__
```

```
  glutInitDisplayMode( GLUT_3_2_CORE_PROFILE | GLUT_DOUBLE | GLUT_RGB |
    GLUT_DEPTH);
#else
  glutInitContextVersion(3,1);
  glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
#endif
  glutInitWindowSize(width, height);
  glutCreateWindow(title);
#ifndef __APPLE__
  glewExperimental = GL_TRUE;
  GLenum err = glewInit() ;
  if (GLEW_OK != err) {
    std::cerr << "Error: " << glewGetErrorString(err) << std::endl;
  }
#endif
  std::cout << "OpenGL Version: " << glGetString(GL_VERSION) << std::endl;
  // END CREATE WINDOW

  initialize();

  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);

  glutMainLoop();
  return 0;
}
```

which requires an additional `initialize()` step, and a set of global variables:

Code 3.2 `HelloSquare.cpp initialize()`

```
void initialize(void){
    printHelp(); // use another function to print the message "Press ESC to
    quit"
    glClearColor (0.0, 0.2, 0.5, 0.0); // background color
    glViewport(0,0,width,height);
    ...
}
```

Code 3.3 `HelloSquare.cpp` headers and global variables

```
#include <iostream>
#ifdef __APPLE__
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
#else
#include <GL/glew.h>
#include <GL/glut.h>
#endif
// Use of degrees is deprecated. Use radians for GLM functions
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>

static const int width = 500;
static const int height = 500;
```

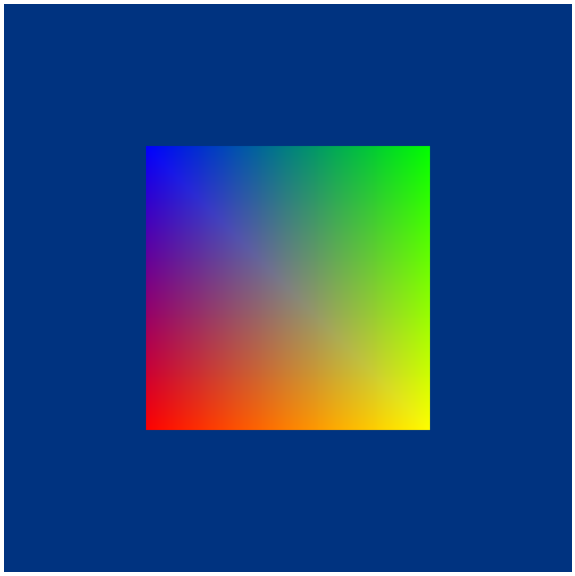


Figure 3.6 The result of Section 3.3: a colorful square.

```
15 static const char* title = "Hello square";
16
17 static GLuint square_vao;  // to be set as the geometry spreadsheet
18 static GLuint buffers[3];  // to be set as buffer objects
19 static GLuint program;     // to be set as the shader program
```

Note that we have declared a few `GLuint` objects. These unsigned integers are the index or the handle to objects that we will assign later. One will be the geometry spreadsheet (vertex array object) describing our geometry, an array of three will be the buffer objects, and the other will become the shader program.

3.3.1 Setup the square geometry

Let us say that in the final normalized screen coordinate $\mathbb{V}_{2D} = [-1, 1] \times [-1, 1]$, we want to draw a square occupying $[-1/2, 1/2] \times [-1/2, 1/2]$. This square can described by the following set of vertices and connectivity.

- There are 4 vertices, indexed by $0, 1, 2, 3$, with positions $p_0 = (-1/2, -1/2)$, $p_1 = (1/2, -1/2)$, $p_2 = (1/2, 1/2)$, $p_3 = (-1/2, 1/2)$.
- There are 2 triangles, attaching to vertices $(0, 1, 3)$ and $(2, 3, 1)$ respectively.

To make the example more interesting, let the 0th vertex be color red (RGB= $(1, 0, 0)$), 1st vertex be yellow (RGB= $(1, 1, 0)$), 2nd vertex be green (RGB= $(0, 1, 0)$) and the 3rd vertex be blue (RGB= $(0, 0, 1)$).

Geometry spreadsheet

The above description for the geometry is summarized in the following spreadsheet.

(a) Vertex attributes

	position		color		
	x	y	r	g	b
0	-0.5	-0.5	1.0	0.0	0.0
1	0.5	-0.5	1.0	1.0	0.0
2	0.5	0.5	0.0	1.0	0.0
3	-0.5	0.5	0.0	0.0	1.0

(b) Connectivity

	incident vertices		
0	0	1	3
1	2	3	1

Table 3.1 Geometry spreadsheet for a colorful square

Let us create these arrays in our C++ code:

Code 3.4 `HelloSquare.cpp initialize()`

```
void initialize(void){
  printHelp(); // use another function to print message
  glClearColor (0.0, 0.2, 0.5, 0.0); // background color
  glViewport(0,0,width,height);

  // define square
  //  blue       green
  //   pt3-----pt2
  //    |  \    |
  //    |    \  |
  //   pt0-----pt1
  // red          yellow
  GLfloat positions[] = {
    -0.5f,-0.5f, // pt0
     0.5f,-0.5f, // pt1
     0.5f, 0.5f, // pt2
    -0.5f, 0.5f  // pt3
  };
  GLfloat colors[] = {
    1.0f, 0.0f, 0.0f, // pt0: red
    1.0f, 1.0f, 0.0f, // pt1: yellow
    0.0f, 1.0f, 0.0f, // pt2: green
    0.0f, 0.0f, 1.0f  // pt3: blue
  };
  GLuint inds[] = { // vertex indices of each triangle
    0, 1, 3, // first triangle
    2, 3, 1  // second triangle
  };
  ...
}
```

Writing data into buffers

Next, we record this data into OpenGL's geometry spreadsheet – a vertex array object. Create *one* vertex array object and *three* buffers; then we activate this geometry spreadsheet (using `glBindVertexArray`):

```
void initialize(void){
  ...
```

```
    glGenVertexArrays(1, &square_vao);
    glGenBuffers(3, buffers);

    glBindVertexArray(square_vao); // activate square_vao
    // the setup written here will be remembered by square_vao
      ...
    glBindVertexArray(0); // deactivate to prevent further modification
    ...
}
```

Now, under the context where `square_vao` is activated, we will do the following. We will write `positions` into the 0th buffer and set it as the 0th attribute; and similarly for other attributes. This is achieved by "inserting" (`glBindBuffer`) the "cassette tape" (our buffer object) into the machine `GL_ARRAY_BUFFER`. Then we can type the numbers from our C++ array `positions` into the `GL_ARRAY_BUFFER` machine (`glBufferData`).

The connectivity data is written using the `GL_ELEMENT_ARRAY_BUFFER` machine instead of `GL_ARRAY_BUFFER`.

Code 3.5 Vertex array object setup

```
1  void initialize(void){
2    ...
3    glBindVertexArray(square_vao); // activate square_vao
4  // the setup written here will be remembered by square_vao
5
6      // Insert buffers[0] to the GL_ARRAY_BUFFER machine
7      glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
8
9      // Type the position data into the GL_ARRAY_BUFFER machine
10     glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions,
     GL_STATIC_DRAW);
11
12     // Configure the 0th attribute from the current buffer
13     glEnableVertexAttribArray(0);
14     glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,(void*)0);
15
16     // Insert buffers[1] to GL_ARRAY_BUFFER, which will automatically eject
     the previous buffers[0]
17     glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
18
19     // Type the color data into the GL_ARRAY_BUFFER machine.
20     glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
21
22     // Configure the 1st attribute from the current buffer
23     glEnableVertexAttribArray(1);
24     glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,0,(void*)0);
25
26     // Write the connectivity to buffers[2] using the GL_ELEMENT_ARRAY_BUFFER
      machine.
27     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
28     glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds, GL_STATIC_DRAW)
     ;
29
30   glBindVertexArray(0); // deactivate to prevent further modification
```

```
31      ...
32  }
```

The option GL_STATIC_DRAW is suitable for the case when the buffer's data is only set once. (If the buffer's data is frequently modified, use GL_DYNAMIC_DRAW. If the buffer will only be modified a few times, use GL_STREAM_DRAW.)

Vertex attribute formatting

The two functions

$$\texttt{glEnableVertexAttribArray}(\langle k\text{-th attribute}\rangle)$$

and

$$\texttt{glVertexAttribPointer}(\ k\ ,\ m\ ,\ \langle type\rangle\ ,\ \texttt{GL\_FALSE},\ s\ ,\ p\ )$$

specify the k-th attribute as a portion of the buffer currently in the GL_ARRAY_BUFFER machine. Here,

- $k \in \{0, 1, 2, \ldots\}$ is the index of the attribute;
- $m \in \{1, 2, 3, \ldots\}$ is the number of components;
- $\langle type\rangle$ is the type of each component;
- The slot where we put GL_FALSE is an option for casting $\{0, \ldots, 255\}$ integers to floats in the interval $[0, 1]$;
- $s \in \{0, 1, 2, \ldots\}$ is the byte stride;
- $p \in \{0, 1, 2, \ldots\} \subset$ (void*) is the byte offset;[2]

describes that the linear stream of bytes in our buffer should be viewed as a table (row-major order) with s (bytes) as the width (number of columns); and the portion of the buffer we select are $(m \cdot \texttt{sizeof}(\langle type\rangle))$ many columns starting from the p-th column. This stride-offset selection is useful when the buffer is an array alternating among many attributes, which may be the case when the array is read from a text file.

If $s = 0$ and $p = 0$, which is the case in our example, then it is equivalent to "select all," that is $s = m \cdot \texttt{sizeof}(\langle type\rangle)$ and $p = 0$.

3.3.2 Shader setup

Now, we setup the shaders. We write the source code for the vertex shader and the fragment shader in GLSL. GLSL is a C-like language that has many mathematical types pre-defined.

Before we explain how to write the shader source code in our C++ file, let us just look at the GLSL code alone.

GLSL basics

In either the vertex or the fragment shader, the first line of code is always a statement of the GLSL version. We will not use versions smaller than 330.

Next we declare the input and output using the keywords in and out. One can also declare uniform variables using the keyword uniform.

[2] There is no particular reason why the stride is an integer and the offset is a pointer other than convention.

The input of the vertex shader comes from the geometry spreadsheet. The input of the fragment shader comes from the output of the vertex shader. The output of the fragment shader goes to the frame buffer. So, these shaders are like Lego pieces. The in and out variables should match exactly. To organize the locations of these in and out plugs, we can use the `layout(location=...)` in front of the `in`, `out`, `uniform` keywords.

Sometimes the `layout` qualifier is not mandatory. For example, if the output variable name of the vertex shader is chosen exactly the same as the input variable name of the fragment shader, then they will automatically match.

One last thing is that the vertex shader must set a GLSL-recognized variable `gl_Position`, which is a `vec4`. This is the position of the vertex in the normalized device coordinate. For reasons we will dive deeper in the future, it is a 4D vector with the last component set as 1.

That is about it for GLSL. The following are a basic vertex shader and a basic fragment shader.

Code 3.6 Vertex shader

```
#version 330 core
// The inputs of the vertex shader are consistent with the vertex attributes in
     the geometry spreadsheet
layout(location = 0) in vec2 pos;
layout(location = 1) in vec3 color;

out vec3 vertexcolor;

void main(){
    gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f );
    vertexcolor = color;
}
```

Code 3.7 Fragment shader

```
#version 330 core
// The inputs to the fragment shader are the outputs variables of the vertex
    shader with the matching names
in vec3 vertexcolor;

// Output the frag color
out vec4 fragColor;

void main(){
    fragColor = vec4(vertexcolor, 1.0f);
}
```

Compile shaders

Back to our `initialize()` in the C++ file `HelloSquare.cpp`. We write the above source code in `char*` string.

Code 3.8 Shader source code

```
  void initialize(void){
    ...
    glBindVertexArray(square_vao);
      ... // setup square geometry (previous section)
    glBindVertexArray(0);

    const char* vertShaderSrc = R"(
      #version 330 core

      layout(location = 0) in vec2 pos;
      layout(location = 1) in vec3 color;

      out vec3 vertexcolor;

      void main(){
          gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f );
          vertexcolor = color;
      }
)";
    const char* fragShaderSrc = R"(
      #version 330 core

      in vec3 vertexcolor;

      // Output the frag color
      out vec4 fragColor;

      void main(){
          fragColor = vec4(vertexcolor, 1.0f);
      }
)";
    ...
  }
```

Tip The C++11 feature `R"(...)"` is the **raw string literal**. It is useful for multiline strings. To use it, make sure the compiler is set to support C++11. For example, in makefile, add `-std=c++11` to the `CFLAGS`. You can also tell an IDE to compile with C++11.

You may also just write multiple lines of classical C strings with `"...;\n"` in every line.

Now, we need to type the source code on the "official sheets of paper." Create shader papers and type our character arrays on them using the "typewriter" called `glShaderSource`. After that, we compile the shaders using the "translator" called `glCompileShader`. Then, we link the compiled shader to produce a shader program. Think of this final step as attaching the compiled shader to a blank book called program, and then printing the binaries onto the program.

Code 3.9 Compile and link shaders

```
1  void initialize(void){
2    ...
```

```
    const char* vertShaderSrc = R"(...)";
    const char* fragShaderSrc = R"(...)";

    // The "official sheet of paper" to write the shader code
    GLuint vs = glCreateShader(GL_VERTEX_SHADER);
    GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(vs, 1, &vertShaderSrc, NULL); // Type our code onto vs
    glShaderSource(fs, 1, &fragShaderSrc, NULL); //           and onto fs
    glCompileShader(vs); // Compile shaders.
    glCompileShader(fs); // Now vs and fs are binaries.

    // program has been defined as a global variable
    program = glCreateProgram();
    glAttachShader(program, vs);
    glAttachShader(program, fs);
    glLinkProgram(program); // Now vs and fs are linked, producing a program.
    glDeleteShader(vs); // clean the pre-linked intermediate binaries
    glDeleteShader(fs);
}
```

This is the end of `initialize()`.

3.3.3 The draw command

We are ready to draw the square. In the display callback function, we write:

Code 3.10 Draw command

```
void display(void){
  glClear(GL_COLOR_BUFFER_BIT);

  // BEGIN draw square
  glBindVertexArray(square_vao);
  glUseProgram(program);
  glDrawElements(GL_TRIANGLES,
                 6, // length of the array inds[]
                 GL_UNSIGNED_INT,0);
  // END draw square

  glutSwapBuffers();
  glFlush();
}
```

To draw the square, we first activate the geometry spreadsheet that describes the square. We also activate the shader program. Then, as we call the `glDrawElements(GL_TRIANGLES,...)` function, the pipeline runs the routine. In particular, it grabs whatever data that is in the geometry spreadsheet, feeds the data into the GPU who runs the shader program.

The arguments of the function `glDrawElements(GL_TRIANGLES,...)` are essentially the length in the array to draw from the element array (the connectivity data). There are six entries in the connectivity table, and each entry is of the type `GL_UNSIGNED_BYTE`.

Later, we will write a more structured and moduled code. This number 6 for the number of connectivities will be more appropriately represented by a class/struct member owned by the geometry.

Now, the program should produce Figure 3.6.

3.4 Rasterization as an interpolator

It is worth noting that, before rasterization, such as in the per-vertex context of the vertex shader, there are only four colors, one for each vertex. After rasterization, there will be as many different colors as the number of pixels covering the square (Figure 3.6). The values of the fragment attributes are **linearly interpolated** from the vertices within each triangle.

We will save the mathematical detail to a later chapter. In short, if a fragment with a screen coordinate $(x, y) \in \mathbb{V}_{2D}$ is in a triangle with vertices at $(x_0, y_0), (x_1, y_1), (x_2, y_2) \in \mathbb{R}^2$, then there is unique set of coefficients

$$0 \leq \lambda_0, \lambda_1, \lambda_2 \leq 1, \quad \lambda_0 + \lambda_1 + \lambda_1 = 1, \tag{3.4}$$

such that

$$\begin{bmatrix} x \\ y \end{bmatrix} = \lambda_0 \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \lambda_1 \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \lambda_2 \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}. \tag{3.5}$$

These λ coefficients are called **barycentric coordinates**. We simply use these coefficients to perform the linear combination. If C_0, C_1, C_2 are the vertex attributes sitting on the vertices, then the rasterizer will assign

$$C = \lambda_0 C_0 + \lambda_1 C_1 + \lambda_2 C_2 \tag{3.6}$$

for the fragment at (x, y).

3.5 Drawing a circle by the shader

In this section, we practice passing uniform variables to the shaders. The goal is to create a picture like Figure 3.7.

To do so, we will primarily modify the fragment shader. In the fragment shader, we will use an `if` condition to check if the fragment is in the circle region of the screen. If so, then we will paint the fragment by a constant color. This color is passed in from the C++ code. We will also let the circle radius be one of the uniform variables passed in from the C++ end.

Then, the set of uniform variables become the **parameter interface** for our shaders. We can easily change the color and the radius of the circle without changing or recompiling the GLSL code.

3.5.1 Adding uniform variables

The new GLSL code are as follows.

Code 3.11 Vertex shader

```
1 # version 330 core
2 layout (location = 0) in vec2 pos;
3 layout (location = 1) in vec3 color;
4 out vec3 vertexcolor;
```

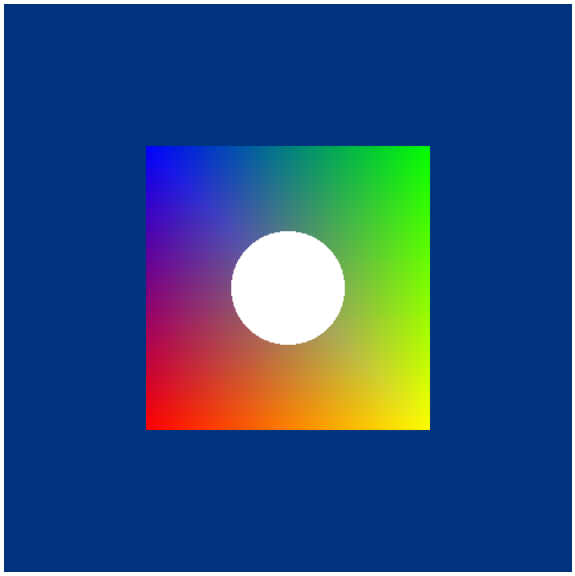


Figure 3.7 The result of Section 3.5.

```
out vec2 vertexpos;          // new
void main(){
    gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f );
    vertexcolor = color;
    vertexpos = pos;         // new
}
```

Code 3.12 Fragment shader

```
#version 330 core

in vec3 vertexcolor;
in vec2 vertexpos;            // new
uniform float circleradius;   // new
uniform vec3 circlecolor;     // new

out vec4 fragColor;

void main (void){
    fragColor = vec4(vertexcolor, 1.0f);
    if (length(vertexpos)<circleradius){      // new
        fragColor = vec4(circlecolor, 1.0f); // new
    }                                         // new
}
```

Note that we have added a new vertex shader output (and correspondingly a fragment shader input) called `vec2 vertexpos`. The reason for doing so is that we need to access the geometric position of the fragment in the fragment shader.

We have also added two uniform variables as our parameter interface.

Now, in the C++ code, we add four global variables:

Code 3.13 `HelloSquare.cpp` global variables

```
...
static const int width = 500;
static const int height = 500;
static const char* title = "Hello square";

static GLuint square_vao;
static GLuint buffers[3];
static GLuint program;

static GLfloat myradius = 0.2;    // new
static GLuint circleradius_loc;   // new
static glm::vec3 mycolor = glm::vec3(1.0,1.0,1.0); //white  // new
static GLuint circlecolor_loc;    // new
...
```

`myradius` and `mycolor` are the values we want to assign to the uniform variables. The `GLuint (...)_loc` are going to be set as the layout location of the uniform variables.

We set the layout location of the uniform variables in the initialization (after the shader compilation).

Code 3.14 Find uniform locations

```
void initialize(void){
  ...
  // Setup geometry
  ...
  // Setup shader
  ...
  // Get Uniform locations
  circleradius_loc = glGetUniformLocation( program, "circleradius" );
  circlecolor_loc = glGetUniformLocation( program, "circlecolor" );
}
```

Finally, at render time, we set the uniform variables:

Code 3.15 Set uniform variable values

```
void display(void){
  glClear(GL_COLOR_BUFFER_BIT);

  glBindVertexArray(square_vao);
  glUseProgram(program);
  glUniform1f(circleradius_loc, myradius ); // new
  glUniform3f(circlecolor_loc, mycolor[0], mycolor[1], mycolor[2] ); // new
  glDrawElements(GL_TRIANGLES,6,GL_UNSIGNED_INT,0);

  glutSwapBuffers();
  glFlush();
}
```

Now, the code should produce Figure 3.7.

4. Modularizing OpenGL Code

Mathematics is the art of giving
the same name to different things.

Henri Poincaré, 1908

In the last chapter, we learned about the basics of OpenGL code. The rendering pipeline converts geometry to pixel color. The geometry is represented by a geometry spreadsheet (vertex array object) with data stored in several buffers. In particular, there are many variables involved just for representing the geometry. A shader program is compiled from a few GLSL source code, which also involves many objects. If the program uses uniform variables, our code will need to have more variables to keep track of those uniform variable locations.

In this chapter, we organize these variables into modules.

4.1 Geometry

A geometry in a rendering pipeline is an object to be drawn. In particular, a geometry contains

- the information that is to be read by the vertex shader; (*i.e.* the geometry spreadsheet, also known as the vertex array, containing the vertex attributes and connectivity);
- the information needed by the `glDrawElements` command.

Setting up the vertex array object (geometry spreadsheet) require buffer objects for storing the actual underlying data. In most cases, different geometries use different underlying data (unless one does geometry instancing), these buffers should be owned by the geometry. When there is a clear ownership (here the buffer-geometry ownership) we let the owned object be a class member of the owner.

4.1.1 Geometry class

We write a class for geometric objects. In the example we give below, it is a class containing only class members without class methods. It is to be made into subclasses (derived classes) that may contain class methods.

For example, we will later write a class called "Square," which is a geometry, and thus a subclass of the Geometry class.

The Geometry class should contain all informations to draw the geometric object. To draw an object `obj` of this Geometry class, we call

Code 4.1 Inside `display()`

```
glBindVertexArray(obj.vao);
glDrawElements(obj.mode, obj.count, obj.type, 0);
```

which should explain the purpose of those class members.

The class is simple enough, so we will just code it in a header file.

Code 4.2 `Geometry.h` Geometry class

```
#include <vector>

#ifndef __GEOMETRY_H__
#define __GEOMETRY_H__

class Geometry {
public:
    GLenum mode = GL_TRIANGLES; // the cookboook for glDrawElements
    int count; // number of elements to draw
    GLenum type = GL_UNSIGNED_INT; // type of the index array
    GLuint vao; // vertex array object a.k.a. geometry spreadsheet
    std::vector<GLuint> buffers; // data storage
};

#endif
```

Note that there is an additional class member, which is an array of buffers.

Tip The array of buffers may have various different array sizes (number of buffers) depending on the object. Instead of manually allocating a dynamic array, we use the `std::vector` template. `std::vector` manages the memory allocation for us regardless of the indefinite array lengths. If we would manually allocate the array, we need to implement a destructor to free the allocated memory.

We can call the `std::vector` function `.data()` to get the underlying array (pointer to the 0th element).

Although we use `std::vector` to manage the array, the entries of the array are actually `GLuint` indices. In a way, the array of buffers in the Geometry class is an array of pointers pointing to the actual buffers. So, it is still a good idea to implement a destructor for the class (as well as the assignment operator and the copy constructor) in a serious application. In the destructor, one would want to manually delete the buffers (with `glDeleteBuffers(...)`).

4.1.2 Square class

A *square* is a special case of a *geometry*.

```
class Square : public Geometry;
```

In addition to a plain geometry, it has a predefined initialization procedure. We implement two class functions:

```
// Initialize a plain square without color
    void init(const glm::vec2 center, const GLfloat sidelength );
// Initialize a square with four colors
    void init(const glm::vec2 center, const GLfloat sidelength,
              const glm::vec3 color0, const glm::vec3 color1,
              const glm::vec3 color2, const glm::vec3 color3);
```

Essentially, we copy the large chunk of code from Section 3.3.1:

Code 4.3 `Square.h` Square class, a subclass of Geometry class

```
/*************************************************
Square is subclass class of Geometry
that represents a 2D square.
*************************************************/
#include "Geometry.h"
#ifndef __SQUARE_H__
#define __SQUARE_H__

class Square : public Geometry {
public:

    // Square without color
    void init(const glm::vec2 center, const GLfloat sidelength ){
        //
        //   pt3-----pt2
        //    |  \    |
        //    |    \  |
        //   pt0-----pt1
        //

        GLfloat positions[] = {
            center.x - 0.5f*sidelength, center.y - 0.5f*sidelength, // pt0
            center.x + 0.5f*sidelength, center.y - 0.5f*sidelength, // pt1
            center.x + 0.5f*sidelength, center.y + 0.5f*sidelength, // pt2
            center.x - 0.5f*sidelength, center.y + 0.5f*sidelength  // pt3
        };
        GLuint inds[] = { // vertex indices of each triangle
            0, 1, 3, // first triangle
            2, 3, 1  // second triangle
        };
        glGenVertexArrays(1, &vao );
        buffers.resize(2); // recall that buffers is std::vector<GLuint>
        glGenBuffers(2, buffers.data());
        glBindVertexArray(vao);

        // 0th attribute: position
        glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions,
```

```
    GL_STATIC_DRAW);
39      glEnableVertexAttribArray(0);
40      glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,(void*)0);
41
42      // indices
43      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[1]);
44      glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds,
    GL_STATIC_DRAW);
45
46      count = sizeof(inds)/sizeof(inds[0]);
47
48      glBindVertexArray(0);
49  }
50
51  // Square with color
52  void init(const glm::vec2 center, const GLfloat sidelength,
53            const glm::vec3 color0, const glm::vec3 color1,
54            const glm::vec3 color2, const glm::vec3 color3){
55      // define square
56      // color3    color2
57      //   pt3-----pt2
58      //    |  \    |
59      //    |    \  |
60      //   pt0-----pt1
61      // color0    color1
62      GLfloat positions[] = {
63          center.x - 0.5f*sidelength, center.y - 0.5f*sidelength, // pt0
64          center.x + 0.5f*sidelength, center.y - 0.5f*sidelength, // pt1
65          center.x + 0.5f*sidelength, center.y + 0.5f*sidelength, // pt2
66          center.x - 0.5f*sidelength, center.y + 0.5f*sidelength  // pt3
67      };
68      GLfloat colors[] = {
69          color0[0], color0[1], color0[2],
70          color1[0], color1[1], color1[2],
71          color2[0], color2[1], color2[2],
72          color3[0], color3[1], color3[2]
73      };
74      GLuint inds[] = { // vertex indices of each triangle
75          0, 1, 3, // first triangle
76          2, 3, 1  // second triangle
77      };
78      glGenVertexArrays(1, &vao );
79      buffers.resize(3); // recall that buffers is std::vector<GLuint>
80      glGenBuffers(3, buffers.data());
81      glBindVertexArray(vao);
82
83      // 0th attribute: position
84      glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
85      glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions,
    GL_STATIC_DRAW);
86      glEnableVertexAttribArray(0);
87      glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,(void*)0);
88
89      // 1st attribute: color
90      glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
```

```
91          glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
92          glEnableVertexAttribArray(1);
93          glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,0,(void*)0);
94
95          // indices
96          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
97          glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds,
        GL_STATIC_DRAW);
98
99          count = sizeof(inds)/sizeof(inds[0]);
100
101         glBindVertexArray(0);
102     }
103 };
104
105 #endif
```

The relationship between Square and Geometry is similar to that Geometry is an abstract class and Square is one example implementation of Geometry. You may wonder whether we could implement these `init` as the constructors for Square, since their purpose is to assign values to the class members. However, it is likely that we will construct these geometries before the OpenGL is initialized (for example, when a Square object is defined as a global variable up front). Before OpenGL is initialized, the manipulation of the OpenGL buffers will lead to segmentation fault.

In the main C++ code, we can declare a Square object:

Code 4.4 `HelloSquare2.cpp` Headers and global variable setup

```
...
#include "Square.h"
static const int width = 500;
static const int height = 500;
static const char* title = "Hello Square!";
static Square square; // <--- declare square as a Square geometry
...
```

initialize with one function call:

Code 4.5 `HelloSquare2.cpp` Inside `initialize()`

```
void initialize(void){
    printHelp();
    glClearColor (0.0, 0.2, 0.5, 0.0); // background color
    glViewport(0,0,width,height);

    // Initialize square
    square.init(glm::vec2(0.0,0.0), 1.0,
                glm::vec3(1.0,0.0,0.0),
                glm::vec3(1.0,1.0,0.0),
                glm::vec3(0.0,1.0,0.0),
                glm::vec3(0.0,0.0,1.0));
```

```
    // Initialize shader
    ...

}
```

and render it intuitively:

Code 4.6 `HelloSquare2.cpp` Inside `display()`

```
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);

    // BEGIN draw square
    glBindVertexArray(square.vao);
    glDrawElements(square.mode, square.count, square.type, 0 );
    // END draw square

    glutSwapBuffers();
    glFlush();
}
```

4.2 Shader

We encapsulate the shader program, the intermediate objects during the compilation, and all the uniform variables and their locations into a class called Shader.

Tip Uniform variables and their locations form a *parameter interface* for the shader. So they should be designed as members of a user's customized shader as a subclass of shader.

The two nontrivial features for the Shader class are

- the ability of reading external files for the GLSL source code;
- the ability of printing error message if the shader compilation fails.

Code 4.7 `Shader.h` The header file for the Shader class.

```
1  #ifndef __SHADER_H__
2  #define __SHADER_H__
3
4  class Shader {
5  private:
6      GLuint vertexshader;      // intermediate shader object
7      GLuint fragmentshader;    //    before the linker stage
8      GLint compiled_vs = 0;   // compile status
9      GLint compiled_fs = 0;   // compile status
10     GLint linked = 0;        // link status
11 public:
12     GLuint program;  // the shader program
13     std::string vertexshader_source;   // source code
14     std::string fragmentshader_source; // source code
15
16     void read_source(const char * vertexshader_filename, const char *
       fragmentshader_filename);
```

```
17      void compile();
18      GLint getVertexShaderCompileStatus(){return compiled_vs;}
19      GLint getFragmentShaderCompileStatus(){return compiled_fs;}
20      GLint getLinkStatus(){return linked;}
21
22  private:
23      // Helper functions
24      static std::string textFileRead(const char * filename );
25      static void programerrors(const GLint program);
26      static void shadererrors(const GLint shader);
27  };
28
29  #endif
```

With such a Shader class, we can set up the shader with just a few lines in our main C++ code:

Code 4.8 `HelloSquare2.cpp` Set up the shader.

```
1  ...
2  #include "Shader.h"
3  ...
4  Static Shader shader;
5  ...
6  void initialize(){
7        ...
8        // Initialize square
9        ...
10       // Initialize shader
11       shader.read_source( "shaders/hello.vert", "shaders/hello.frag");
12       shader.compile();
13       glUseProgram(shader.program);
14 }
15 ...
```

4.2.1 Implementation of the Shader class

Code 4.9 `Shader.cpp` The source code of the Shader class.

```
1  #ifdef __APPLE__
2  #include <OpenGL/gl3.h>
3  #include <GLUT/glut.h>
4  #else
5  #include <GL/glew.h>
6  #include <GL/glut.h>
7  #endif
8
9  #include <iostream>
10 #include <fstream>
11 #include <cstring>
12
13 #include "Shader.h"
14
15 using namespace std ;
16
```

```
17 void Shader::read_source(const char * vertexshader_filename, const char *
       fragmentshader_filename)  {
18     vertexshader_source = textFileRead(vertexshader_filename);
19     fragmentshader_source = textFileRead(fragmentshader_filename);
20 }
21 void Shader::compile()  {
22     vertexshader = glCreateShader(GL_VERTEX_SHADER);
23     fragmentshader = glCreateShader(GL_FRAGMENT_SHADER);
24     const GLchar * cstr_vs = vertexshader_source.c_str() ; // convert source to
        const GLchar *
25     const GLchar * cstr_fs = fragmentshader_source.c_str() ;
26     glShaderSource( vertexshader, 1, &cstr_vs, NULL) ;
27     glShaderSource( fragmentshader, 1, &cstr_fs, NULL) ;
28     glCompileShader( vertexshader );
29     glCompileShader( fragmentshader );
30     glGetShaderiv ( vertexshader, GL_COMPILE_STATUS, &compiled_vs ) ;
31     glGetShaderiv ( fragmentshader, GL_COMPILE_STATUS, &compiled_fs ) ;
32     if (!compiled_vs) {
33         cout << "Vertex Shader ";
34         shadererrors( vertexshader ) ;
35         throw 3 ;
36     }
37     if (!compiled_fs) {
38         cout << "Fragment Shader ";
39         shadererrors( fragmentshader ) ;
40         throw 3 ;
41     }
42     program = glCreateProgram() ;
43     glAttachShader(program, vertexshader) ;
44     glAttachShader(program, fragmentshader) ;
45     glLinkProgram(program) ;
46     glGetProgramiv(program, GL_LINK_STATUS, &linked) ;
47     if (linked){
48         glDetachShader( program, vertexshader );
49         glDetachShader( program, fragmentshader );
50         glDeleteShader( vertexshader );
51         glDeleteShader( fragmentshader );
52     }else{
53         programerrors(program) ;
54         throw 4 ;
55     }
56 }
57
58 // Below are helper functions
59
60 string Shader::textFileRead (const char * filename) {
61         string str, ret = "" ;
62         ifstream in ;
63         in.open(filename) ;
64         if (in.is_open()) {
65                 getline (in, str) ;
66                 while (in) {
67                         ret += str + "\n" ;
68                         getline (in, str) ;
69                 }
```

```
                //    cout << "Shader below\n" << ret << "\n" ;
                return ret ;
        }
        else {
                cerr << "Unable to Open File " << filename << "\n" ;
                throw 2 ;
        }
}

void Shader::programerrors (const GLint program) {
        GLint length ;
        GLchar * log ;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &length) ;
        log = new GLchar[length+1] ;
        glGetProgramInfoLog(program, length, &length, log) ;
        cout << "Compile Error, Log Below\n" << log << "\n" ;
        delete [] log ;
}
void Shader::shadererrors (const GLint shader) {
        GLint length ;
        GLchar * log ;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length) ;
        log = new GLchar[length+1] ;
        glGetShaderInfoLog(shader, length, &length, log) ;
        cout << "Compile Error, Log Below\n" << log << "\n" ;
        delete [] log ;
}
```

4.2.2 Subclass of the Shader class

In the example of Section 3.5, the shader program has two uniform variables: the radius of a circle, and the color of that circle. In this situation, we make a subclass of the shader class.

We call the subclass CircleShader, because as shown in Section 3.5 the shader is capable of drawing a circle.

In the main C++ code, we set up the global variables and define the CircleShader class:

Code 4.10 `HelloSquare2.cpp` Headers and global variable setup

```
...
#include "Shader.h"
#include "Square.h"

static const int width = 500;
static const int height = 500;
static const char* title = "Hello Square!";
static Square square;
class CircleShader : public Shader {
public:
    GLfloat radius = 0.2; GLuint radius_loc;
    glm::vec3 color = glm::vec3(1.0,1.0,1.0); GLuint color_loc;
    void initUniforms(){
        radius_loc = glGetUniformLocation( program, "circleradius" );
```

```
        color_loc = glGetUniformLocation( program, "circlecolor" );
    }
    void setUniforms(){
        glUniform1f(radius_loc, radius);
        glUniform3f(color_loc, color[0],color[1],color[2]);
    }
};
static CircleShader shader;
...
```

The full code for the subroutine `initize()` is now:

Code 4.11 `HelloSquare2.cpp`

```
...
void initialize(void){
    printHelp();
    glClearColor (0.0, 0.2, 0.5, 0.0); // background color
    glViewport(0,0,width,height);

    // Initialize square
    square.init(glm::vec2(0.0,0.0), 1.0,
                glm::vec3(1.0,0.0,0.0),
                glm::vec3(1.0,1.0,0.0),
                glm::vec3(0.0,1.0,0.0),
                glm::vec3(0.0,0.0,1.0));
    // Initialize shader
    shader.read_source( "shaders/hello.vert", "shaders/hello.frag");
    shader.compile();
    glUseProgram(shader.program);
    shader.initUniforms(); // <----New
    shader.setUniforms();  // <----New
}
...
```

Whenever we want to modify the value of the uniform variables, we can just call `shader.setUniforms()`. In particular, `setUniforms()` can also be called in the run time `display()`.

4.3 Screenshot (optional)

In this section, we give a module for taking a screenshot for your window. Having this screenshot capability is optional, as you can use other screen capture tool from your operating system.

Code 4.12 `Screenshot.h` A single header file defining the Screenshot class.

```
/**
 The Screenshot class allows us to save a screenshot for our OpenGL program.
 Specifically, it reads the pixels data from the GL_FRONT buffer and
 writes a .png image file.

 The (private) class members of a Screenshot object are the width, height for
```

```
    the image, and the raw pixel data. The pixel data should be an array of
    BTYEs. Instead of manually allocating the array, we store the pixel data
    using std::vector<BYTE>.  Note that for pixels_ of type std::vector<BYTE>,
    pixels_.data() gives us exactly the desired raw array of BYTEs.  Using std::
    vector to encapsulate the array, we don't need to worry about writing an
    appropriate destructor for the class.
 **/

#ifndef __SCREENSHOT_H__
#define __SCREENSHOT_H__
#include <FreeImage.h>
#include <vector>
#include <iostream>

class Screenshot {

public:
    Screenshot(int width, int height) :
    width_(width), height_(height) {
        pixels_.resize(width_ * height_ * 3);
    }
    void save(const char* filename){
        glReadBuffer(GL_FRONT);
        glReadPixels(0,0,width_,height_,GL_BGR,GL_UNSIGNED_BYTE,pixels_.data()
    );

        FIBITMAP *img = FreeImage_ConvertFromRawBits(pixels_.data(), width_,
    height_, width_ * 3, 24, 0xFF0000, 0x00FF00, 0x0000FF, false);

        std::cout << "Saving screenshot: " << filename << std::endl;

        FreeImage_Save(FIF_PNG, img, filename, 0);
    }

private:
    int width_;    // Window width
    int height_;   // Window height
    std::vector<BYTE> pixels_; // raw pixel data
};

#endif
```

The following is an example of using the Screenshot class. In our program, pressing the key "o" produces a screenshot named "`text.png`":

Code 4.13 `HelloSquare2.cpp`

```
...
#include "Screenshot.h"
...
void saveScreenShot(void){
    int currentwidth = glutGet(GLUT_WINDOW_WIDTH);
    int currentheight = glutGet(GLUT_WINDOW_HEIGHT);
    Screenshot imag = Screenshot(currentwidth,currentheight);
    imag.save("screenshot_HelloSquare2.png");
```

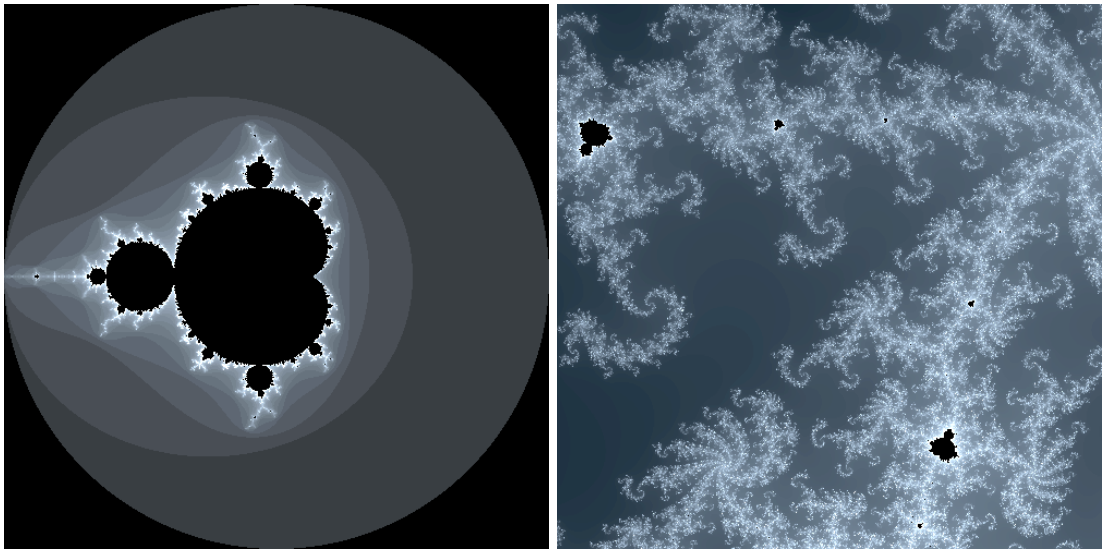


Figure 4.1 The Mandelbrot fractal as a whole (left); a zoomed-in shot of the fractal (right).

```
9  }
10
11 void keyboard(unsigned char key, int x, int y){
12     switch(key){
13         case 27: // Escape to quit
14             exit(0);
15             break;
16         case 'h': // print help
17             printHelp();
18             break;
19         case 'o': // save screenshot
20             saveScreenShot();
21             break;
22         default:
23             glutPostRedisplay();
24             break;
25     }
26 }
27 ...
```

4.4 Mandelbrot fractal shader

This section contains an exercise where you will write your own shader (a subclass of the Shader class) for cool visualizations. Specifically, we will render the **Mandelbrot set**.[1]

The Mandelbrot set is a set in the complex plane. It can be defined with a very simple mathematical procedure, but the resulting picture of the set is visually stunning.

Mandelbrot set is a **fractal**. For a fractal, you can endlessly zoom into the picture and find a similar copy of the picture at a larger scale. For Mandelbrot set, the

[1] https://en.wikipedia.org/wiki/Mandelbrot_set

zoomed in versions of the picture are all slightly different depending on the scale and depending on where you zoom in. You can find infinitely many beautiful patterns by exploring different parts of the Mandelbrot set.

The Mandelbrot set is defined by the following procedure. Consider the iteration

Input: $c \in \mathbb{C}$;
1: $z_0 = 0 \in \mathbb{C}$;
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: $\quad z_{k+1} = z_k^2 + c$;
4: **end for**

For each fixed complex number c, we have a simple nonlinear iteration (the $(\cdot)^2$ in Line 3 makes it nonlinear). It turns out that for such a simple iteration, it is very difficult to predict where the iterant z_k is going. To demonstrate this quality, we visualize the behavior of the iteration depending on c. For c ranging over a square region in the complex plane, we generate a square picture, one value of c per pixel.

A standard aspect of the iteration behavior to quantify is "whether the iteration blows up." If c has a big magnitude, say $c = 10\mathrm{i}$, then we can see that the iterant seems to grow unboundedly. If c has a small magnitude, say $c = 0.1$, then the iteration stays bounded.

Definition 4.1 The **Mandelbrot set** is defined by

$$M = \{c \in \mathbb{C} \mid \text{the above iteration stays bounded.}\} \tag{4.1}$$

If we observe $|z_k| \geq 2$ during the iteration, then the iteration is guaranteed to blow up. The circle centered at the origin with radius 2 is the circle of no return.

So, to make the visualization of the Mandelbrot set more informative, we color the pixel according to the number of iteration k it takes for $|z_k|$ to exceed 2. If the pixel is in the Mandelbrot set (in which case the iterant never has a norm exceeding 2) then we just color it with a default color.

In other words, we define a function

$$I \colon \mathbb{C} \to \{0\} \cup \mathbb{N} \tag{4.2}$$

with

Input: $c \in \mathbb{C}, n_{\max} \in \mathbb{N}$;
1: $I(c) \leftarrow 0$;
2: $z \leftarrow 0 \in \mathbb{C}$;
3: **for** $k = 0, 1, 2, \ldots, n_{\max} - 1$ **do**
4: $\quad z \leftarrow z_k^2 + c$;
5: $\quad$ **if** $|z| > 2$ **then**
6: $\quad\quad I(c) \leftarrow k$;
7: $\quad\quad$ **break**;
8: $\quad$ **end if**
9: **end for**

So $I(c)$ is 0 if c is in the Mandelbrot set (checked up to the max iteration number $n_{\max}$) and otherwise $I(c)$ is the number of iterations it takes for z to pass the the

circle of no return.

Exercise 4.1 Build a Mandelbrot shader (*e.g.* as a subclass of the Shader class). In particular, use a fragment shader to visualize $I(c)$. To do so, put a square covering the viewport as the canvas. You may want to let the vertex shader pass the position variable down to the fragment shader. In the fragment shader, each position $P \in \mathbb{V}_{2D} = [-1,1] \times [-1,1]$ corresponds to a complex number c by

$$c = \texttt{center} + \texttt{zoom} \cdot P \tag{4.3}$$

where the 2D vector (complex number) `center`$\in \mathbb{C}$ and the positive scalar `zoom`$\in \mathbb{R}_{>0}$ are uniform variables. Also set the max iteration $n_{\max}$ as a uniform variable. You can control the center and the zoom of our visualization region in the complex plane by implementing keyboard events.

You may want to write a customized function `vec2 cprod(const vec2 z1, const vec2 z2)` before the `void main(void)` in the fragment shader for the complex number multiplications.

Assign the fragment color according to the value of $I(c)$. You can design your own functions that would map the value of $I(c)$ (casted to float, divided by max iteration) to each channel of the color.

1. Take a screenshot when `center` $= (0,0)$ (*i.e.* $0 + 0\mathrm{i}$) and `zoom` $= 2$ so that we see the entire Mandelbrot set.
2. Zoom in and navigate to a favorite location and take a screenshot.

■

II Linear Algebra and Projective Geometry

5. Linear Algebra

Algebra is but written geometry and geometry is but figured algebra.

Sophie Germain, 1821

In this sequence of chapters, we will study an important component of 3D computer graphics – transformations. The goal of transformations is to set the correct vertex position in the normalized device coordinate, so that the rasterized image reproduces a realistic depiction of the 3D object. The underlying mathematical principles are based on **projective geometry**. Linear algebra allows us to formulate these projective geometric concepts concretely. We will cover linear algebra and linear transformations in this chapter.

5.1 Matrix algebra

A standard course on linear algebra is typically a study of **matrices**. As we will see later, a matrix can represent a linear transformation when a basis or a coordinate system for the space is established. Before the latter geometric structure (namely basis) for the space is given, a matrix is a purely algebraic entity. Let us talk about matrices algebraically.

A matrix is a rectangular table of objects, where the objects may be numbers,

symbols, or submatrices, *etc.*:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \text{ is called an } m\text{-by-}n \text{ matrix.} \tag{5.1}$$

When the objects (for the matrix entries) have the algebraic structure of multiplication and addition, then the corresponding matrices are also equipped with additions and multiplications in the following way.

When $\mathbf{A}$ and $\mathbf{B}$ are both m-by-n, then $\mathbf{A} + \mathbf{B}$ is also an m-by-n matrix given by

$$\mathbf{C} = \mathbf{A} + \mathbf{B}, \quad \text{where } c_{ij} = a_{ij} + b_{ij} \text{ for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n. \tag{5.2}$$

On the other hand, the matrix multiplication $\mathbf{AB}$ requires $\mathbf{A}$ being m-by-n and $\mathbf{B}$ being n-by-ℓ, *i.e.* the number of columns in $\mathbf{A}$ must equal the number of rows in $\mathbf{B}$:

$$\mathbf{C} = \mathbf{AB} \text{ is } m\text{-by-}\ell, \quad \text{where } c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{5.3}$$

Depending on the context, c_{ik} is either interpreted as a **linear combination** of objects $a_{i1}, \ldots, a_{in}$ with coefficients $b_{1k}, \ldots, b_{nk}$, or interpreted as a linear combination of objects $b_{1k}, \ldots, b_{nk}$ with coefficients $a_{i1}, \ldots, a_{in}$. *Matrix multiplications are shorthands for linear combinations.*

A multiplication operator is called **associative** if a sequence of such multiplication can be evaluated independent of the precedence: $(ab)c = a(bc)$. If the objects in the matrix entries are equipped with an associative multiplication, *then the resulting matrix multiplication is also associative.* That is, $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$ whenever they are allowed to be multiplied.

> **Note 5.1** *In general,* $\mathbf{AB} \neq \mathbf{BA}$. *That is, matrix multiplications are not* **commutative**.

5.1.1 Matrix transposition

The **transpose** of an m-by-n matrix $\mathbf{P}$ is denoted and defined by

$$\mathbf{P}^{\mathsf{T}} = \mathbf{Q}, \quad \text{a } n\text{-by-}m \text{ matrix, where } q_{ij} = p_{ji}^{(\mathsf{T})}. \tag{5.4}$$

That is, the transposition simply rearranges matrix entries by swapping rows and columns. The additional transposition $p_{ji}^{(\mathsf{T})}$ is required when the objects p_{ji}'s are matrices themselves.

> **Note 5.2** *Transposition distributes over matrix multiplications and reverses the order of multiplications:*
>
> $$(\mathbf{AB} \cdots \mathbf{C})^{\mathsf{T}} = \mathbf{C}^{\mathsf{T}} \cdots \mathbf{B}^{\mathsf{T}} \mathbf{A}^{\mathsf{T}}. \tag{5.5}$$

5.1.2 Matrix inversion and linear system

The **identity matrix** is defined as a square matrix with 1's in the diagonal and 0 elsewhere:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{5.6}$$

which has the property that

$$\mathbf{AI} = \mathbf{A}, \quad \mathbf{IB} = \mathbf{B}, \tag{5.7}$$

for any rectangular matrices $\mathbf{A}, \mathbf{B}$ of numbers, as long as the sizes of the matrices match so that matrix multiplications make sense.

For matrices of real or complex numbers, a matrix $\mathbf{Q}$ is said to be the **inverse** of a given matrix $\mathbf{P}$ if they are square matrices and

$$\mathbf{QP} = \mathbf{I}, \quad \text{which in fact also implies } \mathbf{PQ} = \mathbf{I}. \tag{5.8}$$

We denote the inverse matrix by

$$\mathbf{P}^{-1} = \mathbf{Q}. \tag{5.9}$$

Note 5.3 • *Not all square matrices are invertible.*

• *When a list of square matrices* $\mathbf{A}, \mathbf{B}, \ldots, \mathbf{C}$ *are invertible, then the product* $\mathbf{AB}\cdots\mathbf{C}$ *is invertible, and*

$$(\mathbf{AB}\cdots\mathbf{C})^{-1} = \mathbf{C}^{-1}\cdots\mathbf{B}^{-1}\mathbf{A}^{-1}, \tag{5.10}$$

i.e. *the inversion operator* $(\cdot)^{-1}$ *distributes over matrix multiplications and reverses the order.*

The inverse of transposed matrix equals to the transpose of inverted matrix: $(\mathbf{A}^{-1})^{\intercal} = (\mathbf{A}^{\intercal})^{-1}$. *So we may write* $\mathbf{A}^{-\intercal}$ *for the inverse transpose without worrying about the ambiguous precedence.*

The matrix algebra arises naturally in the context of systems of linear equations. A system of linear equations consists of a list of unknown variables $x_1, \ldots, x_n$, coefficients $a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{mn}$, and $b_1, \ldots, b_m$ set up as

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m. \end{cases} \tag{5.11}$$

A system of linear equations as such can be written in terms of matrices

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}, \quad \text{that is, } \mathbf{Ax} = \mathbf{b}. \tag{5.12}$$

Here, $\mathbf{x}$ and $\mathbf{b}$ are single-columned matrices. The unknown variables are solvable if and only if the matrix $\mathbf{A}$ is invertible (which necessarily requires that the number m of equations equals the number n of unknowns). By multiplying $\mathbf{A}^{-1}$ from the left on both sides of $\mathbf{Ax} = \mathbf{b}$, we get

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{5.13}$$

In a course of linear algebra, one would learn about what are the conditions for a matrix to be invertible. If it is not invertible, one studies the rank and nullity of a matrix, the four fundamental subspaces of a matrix, the notion of least-squares solutions of solution, *etc*. We will skip these topics of linear algebra because we will not use them in the chapter. But these keywords are very important in almost all scientific and engineering problems. Be sure to also refresh your linear algebra skills on those topics.

5.2 Geometric and algebraic aspects of vectors

In many contexts in computer science and linear algebra, a vector is often referred to as an ordered list of numbers

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n. \tag{5.14}$$

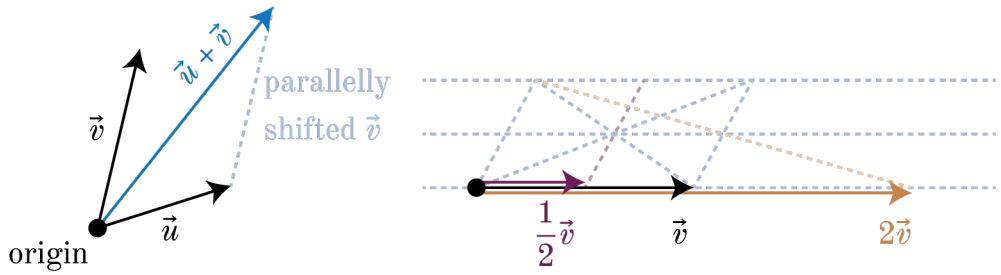
However, here, we want to reserve the name "**vector**" to the geometric entity of an "arrow" that has the notion of *direction* and *magnitude*. Indeed, the array of numbers (5.14) *can have a geometric representation of an arrow*. But this geometric representation requires some artificial and conscious choice of *basis*.

In computer graphics, we constantly change this *basis*, and therefore it is important to clarify the geometry we want to represent and its relationship to the array of numbers that we write in computer code.

5.2.1 Vector space

Definition 5.1 A **real vector space** (also called **real linear space**) V is a set of objects called vectors $\vec{v} \in V$ with the following two operations:

- **addition** $(\vec{u} \in V, \vec{v} \in V) \mapsto \vec{u} + \vec{v}$:
  - Closure: given any $\vec{u}, \vec{v} \in V$, we must also have $\vec{u} + \vec{v} \in V$;
  - Commutative: $\vec{u} + \vec{v} = \vec{v} + \vec{u}$;
  - There is a special vector called the **zero vector** $\vec{0} \in V$ so that $\vec{0} + \vec{v} = \vec{v}$



(a) Vector addition **(b)** Vector scaling

Figure 5.1 A space with an origin and a notion of parallelism is an example of a real vector space.

for all $\vec{v}$;
  - Given any $\vec{v} \in V$, there is a unique vector $-\vec{v} \in V$ so that $\vec{v} + (-\vec{v}) = \vec{0}$.
- **scalar multiplication** (also called **scaling**) $(a \in \mathbb{R}, \vec{v} \in V) \mapsto a\vec{v}$:
  - Closure: given any scalar $a \in \mathbb{R}$ and a vector $\vec{v} \in V$, we must have $a\vec{v} \in V$;
  - Distributive law: $(a+b)\vec{v} = a\vec{v} + b\vec{v}$ and $a(\vec{u}+\vec{v}) = a\vec{u} + a\vec{v}$ for all $a, b \in \mathbb{R}$ and $\vec{u}, \vec{v} \in V$.

▪ **Example 5.1 — Algebraic.** The space $\mathbb{R}^n$ of arrays of real numbers of size n

$$\mathbb{R}^n = \left\{ \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \middle| \, v_1, \ldots, v_n \in \mathbb{R} \right\} \tag{5.15}$$

is a real vector space. The addition and scaling operations are the standard ones. For this algebraic example of a vector space, we write the element in the space with a boldface letter $\mathbf{v} \in \mathbb{R}^n$. ▪

▪ **Example 5.2 — Geometric.** The collection of all arrows based at the origin in a plane (a 3D space, or higher dimensional spaces) is a real vector space. The only requirement of the space, other than having an origin, is that it has a notion of parallelism. As shown in Figure 5.1 (a), we can construct vector addition by connecting a diagonal of the parallelogram formed by the two vectors. As demonstrated in Figure 5.1 (b), by constructing an arbitrary parallelogram with $\vec{v}$ as one side, connecting the diagonals, and by drawing auxiliary parallel lines, we can scale $\vec{v}$ along its extended line dyadically (and thus approximating all real scaling).

One can verify that all the conditions for a real vector space (Definition 5.1) are fulfilled.

Notationwise, for this geometric version of a vector space, we prefer to use an arrow decoration as in "$\vec{v}$" for each element in the vector space. ▪

You may have noticed that the two operations (addition and scaling) are exactly the minimal ingredients for **linear combinations**. An expression

$$a_1\vec{v}_1 + a_2\vec{v}_2 + \ldots + a_k\vec{v}_k, \quad a_1, \ldots, a_k \in \mathbb{R}, \vec{v}_1, \ldots, \vec{v}_k \in V \tag{5.16}$$

is a linear combination of $\vec{v}_1, \ldots, \vec{v}_k$ with coefficients $a_1, \ldots, a_k$.

5.2.2 Basis

Definition 5.2 Let V be a real vector space. A set of vectors $\vec{e}_1, \ldots, \vec{e}_n \in V$ is said to be a **basis** for V if for any vector $\vec{v} \in V$ there exists a unique set of numbers $v_1, \ldots, v_n \in \mathbb{R}$ such that

$$\vec{v} = v_1\vec{e}_1 + v_2\vec{e}_2 + \cdots + v_n\vec{e}_n. \tag{5.17}$$

For a set of vectors to be a basis, it has to consist of enough vectors to *span* the vector space. For example, if V is a 3D space, then we must have at least 3 basis vectors, otherwise we would not be able to represent a vector outside of the lower dimensional line or plane reached by the one or two basis vectors.

Definition 5.3 The **span** of a set of vectors $\vec{e}_1, \ldots, \vec{e}_n \in V$ is the *smallest vector space* containing $\vec{e}_1, \ldots, \vec{e}_n$, denoted by $\text{span}\{\vec{e}_1, \ldots, \vec{e}_n\} \subseteq V$.

For a set of vectors to be a basis, there also should not be too many vectors, otherwise we would loose the uniqueness of the representation (5.17).

Definition 5.4 A set of vectors $\vec{e}_1, \ldots, \vec{e}_n \in V$ are said to be **linearly independent** if

$$a_1\vec{e}_1 + \cdots a_n\vec{e}_n = \vec{0} \quad \text{implies} \quad a_1 = \cdots = a_n = 0. \tag{5.18}$$

One can verify that $\vec{e}_1, \ldots, \vec{e}_n \in V$ is a basis if and only if

- $\text{span}\{e_1, \ldots, e_n\} = V$, and
- $e_1, \ldots, e_n$ are linearly independent.

Note 5.4 *A vector space may not have a basis. For example, an infinite dimensional vector space does not have a basis as defined in Definition 5.2.*

Basis for a vector space is also not unique. There can be many different bases for the same vector space.

A set of basis vectors do not need to be orthogonal to each other. In fact, there is no natural notion of orthogonality between vectors. There is only the notion of parallelism in the space.

Definition 5.5 If a real vector space V has a basis $\vec{e}_1, \ldots, \vec{e}_n$. Then we say V has dimension n.[a]

[a] For this definition to make sense, one has to verify that if there is another set of basis $\vec{f}_1, \ldots, \vec{f}_m$ for the same space, then we must have $m = n$.

▪ **Example 5.3** $\mathbb{R}^n$ is a real vector space with dimension n. ▪

5.2.3 Bridging the geometric and algebraic versions of vectors

Let V be an n-dimensional geometric real vector space, *i.e.* it is a set of arrows (Example 5.2). Then a basis $\vec{e}_1, \ldots \vec{e}_n$ gives us a "bridge" between the geometric

vectors and array of numbers.

For each $\vec{v} \in V$, there is a unique set of numbers $v_1, \ldots, v_n$ so that $\vec{v} = v_1\vec{e}_1 + \cdots + v_n\vec{e}_n$.

Conversely, for each array of n numbers $\mathbf{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$, we have a geometric representation of an arrow given by $\vec{v} = v_1\vec{e}_1 + \cdots + v_n\vec{e}_n$.

Using matrix multiplication (which is the ultimate shorthand for linear combinations), we write the relationship between $\vec{v}$ and $\mathbf{v}$ by

$$\vec{v} = \underbrace{\begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \cdots & \vec{e}_n \end{bmatrix}}_{\vec{\mathbf{e}}^{\mathsf{T}}} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}}_{\mathbf{v}} = \vec{\mathbf{e}}^{\mathsf{T}}\mathbf{v}. \tag{5.19}$$

$\vec{\mathbf{e}}$ is an array of geometric vectors (and thus it is boldfaced and decorated with an arrow), representing the basis, and $\mathbf{v}$ is an array of numbers.

Summary 5.1 A basis $\vec{\mathbf{e}}^{\mathsf{T}}$ turns an array of number $\mathbf{v}$ into a vector $\vec{v}$:

$$\vec{v} = \vec{\mathbf{e}}^{\mathsf{T}}\mathbf{v} \tag{5.20}$$

5.3 Linear transformations

In computer graphics, we will perform many transformations. In graphics, we will see many seemingly non-linear transformations (for example making perspective distortion and translations). But, as we will see in later chapters, through some further tricks all relevant transformations in graphics are all linear. So, let us focus on linear transformations.

Definition 5.6 A linear transformation A on a real vector space V is a map $A\colon V \to V$ that preserves linear combinations. That is,

$$A(c_1\vec{v}_1 + \cdots + c_k\vec{v}_k) = c_1A(\vec{v}_1) + \cdots + c_kA(\vec{v}_k). \tag{5.21}$$

Note 5.5 *A linear transformation must map the origin to the origin* $A(\vec{0}) = \vec{0}$.

A linear transformation can be represented by a matrix when there is a basis. Let $\vec{\mathbf{e}}$ be a basis for V. Now, let us consider $A(\vec{v})$ for a generic vector $\vec{v} \in V$.

$$\begin{aligned} A(\vec{v}) &= A(\vec{\mathbf{e}}^{\mathsf{T}}\mathbf{v}) = A\left(v_1\vec{e}_1 + \cdots v_n\vec{e}_n\right) \\ &= v_1A(\vec{e}_1) + \cdots + v_nA(\vec{e}_n) = \begin{bmatrix} A\vec{e}_1 & \cdots & A\vec{e}_n \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = (A\vec{\mathbf{e}}^{\mathsf{T}})\mathbf{v}. \end{aligned} \tag{5.22}$$

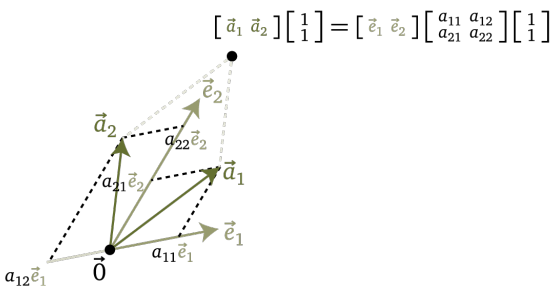


Figure 5.2 A matrix $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ describes the relationship between two bases $\vec{\mathbf{a}}^\intercal = [\,\vec{a}_1\ \vec{a}_2\,]$ and $\vec{\mathbf{e}}^\intercal = [\,\vec{e}_1\ \vec{e}_2\,]$. A vector $\vec{v} = \vec{\mathbf{a}}^\intercal \mathbf{v}$ described by list of coefficients $\mathbf{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ in basis $\vec{\mathbf{a}}$ can also be described by the coefficients $\mathbf{A}\mathbf{v}$ in basis $\vec{\mathbf{e}}$.

Now, each of $A\vec{e}_1, \ldots, A\vec{e}_n$ is once again a vector in V, which can be expanded in terms of the basis $\vec{e}_1, \ldots, \vec{e}_n$:

$$A\vec{e}_k = \vec{a}_k = a_{1k}\vec{e}_1 + \cdots + a_{nk}\vec{e}_n. \tag{5.23}$$

That is,

$$A\vec{\mathbf{e}}^\intercal = \begin{bmatrix} A\vec{e}_1 & \cdots & A\vec{e}_n \end{bmatrix} = \underbrace{\begin{bmatrix} \vec{a}_1 & \cdots & \vec{a}_n \end{bmatrix}}_{\vec{\mathbf{a}}} = \begin{bmatrix} \vec{e}_1 & \cdots & \vec{e}_n \end{bmatrix} \underbrace{\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}}_{\mathbf{A}} \tag{5.24}$$

Summary 5.2 Let $\vec{\mathbf{e}}$ be a basis for a real vector space. Let A be a linear transformation on the vector space. Then $\vec{\mathbf{a}}^\intercal = A\vec{\mathbf{e}}^\intercal$ is called the **transformed basis** and $\mathbf{A}$ is called **matrix representation** of A under the relationship

$$A\vec{\mathbf{e}}^\intercal = \vec{\mathbf{e}}^\intercal \mathbf{A} \tag{5.25}$$
$$= \vec{\mathbf{a}}^\intercal. \tag{5.26}$$

The formula (5.25) is convenient. A linear transformation applied to a vector $A(\vec{v})$:

$$A(\vec{v}) = A\vec{\mathbf{e}}^\intercal \mathbf{v} = \vec{\mathbf{e}}^\intercal \mathbf{A}\mathbf{v} = \vec{\mathbf{e}}^\intercal(\mathbf{A}\mathbf{v}) \tag{5.27}$$

becomes a matrix-vector(array) multiplication between $\mathbf{A}$ and $\mathbf{v}$. That is, we can interpret $A(\vec{v})$ as modifying the coefficients $\mathbf{v}$ of $\vec{v}$ using the matrix multiplication by $\mathbf{A}$

Conversely, given a transformation described by an algebraic matrix-vector multiplication $\mathbf{v} \mapsto \mathbf{A}\mathbf{v}$, we can represent it geometrically by applying a basis

$$\begin{array}{lll} \text{before transformation} & & \text{after transformation} \\ \vec{\mathbf{e}}^\intercal \mathbf{v} & \mapsto & \vec{\mathbf{e}}^\intercal \mathbf{A}\mathbf{v} \\ & & = (A\vec{\mathbf{e}}^\intercal)\mathbf{v} = \vec{\mathbf{a}}^\intercal \mathbf{v}. \end{array} \tag{5.28}$$

We see that before and after the transformation, the coefficients are both $\mathbf{v}$. It is the basis $\vec{\mathbf{e}}$ that has been replaced by $\vec{\mathbf{a}}$.

Summary 5.3 An array of numbers $\mathbf{v}$ can have many different geometric realizations depending on the choice of basis. For example, a basis $\vec{\mathbf{a}}$ realizes $\mathbf{v}$ into $\vec{\mathbf{a}}^{\mathsf{T}}\mathbf{v}$, which can be a different arrow from $\vec{\mathbf{e}}^{\mathsf{T}}\mathbf{v}$ using another basis $\vec{\mathbf{e}}$. The relationship between two different bases is kept track of by an (invertible) square matrix $\mathbf{A}$

$$\vec{\mathbf{a}}^{\mathsf{T}} = \vec{\mathbf{e}}^{\mathsf{T}}\mathbf{A}. \tag{5.29}$$

Tip You can even use the following notation (not a common notation). If $\vec{\mathbf{a}}$ and $\vec{\mathbf{e}}$ are bases, then the matrix $\mathbf{A}$ in $\vec{\mathbf{a}}^{\mathsf{T}} = \vec{\mathbf{e}}^{\mathsf{T}}\mathbf{A}$ is the "ratio"

$$\mathbf{A} = \frac{\vec{\mathbf{a}}^{\mathsf{T}}}{\vec{\mathbf{e}}^{\mathsf{T}}}. \tag{5.30}$$

If we have a vector described by the array $\mathbf{v}$ and the basis $\vec{\mathbf{a}}$, we will know how to write it out in terms of the basis $\vec{\mathbf{e}}$: Just multiply the formula by $\vec{\mathbf{e}}^{\mathsf{T}}$ and then divide by $\vec{\mathbf{e}}^{\mathsf{T}}$

$$\vec{\mathbf{a}}^{\mathsf{T}}\mathbf{v} = \vec{\mathbf{e}}^{\mathsf{T}}\frac{\vec{\mathbf{a}}^{\mathsf{T}}}{\vec{\mathbf{e}}^{\mathsf{T}}}\mathbf{v} = \vec{\mathbf{e}}^{\mathsf{T}}(\mathbf{A}\mathbf{v}). \tag{5.31}$$

Note that

$$\frac{\vec{\mathbf{e}}^{\mathsf{T}}}{\vec{\mathbf{a}}^{\mathsf{T}}} = \left(\frac{\vec{\mathbf{a}}^{\mathsf{T}}}{\vec{\mathbf{e}}^{\mathsf{T}}}\right)^{-1} \tag{5.32}$$

where $(\cdot)^{-1}$ is the matrix inversion.

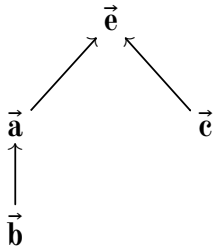
5.4 Transformations in graphics

The transformations in graphics are best understood in terms of Summary 5.3.

Every geometric object are given by numbers encoded in the geometry spreadsheet (vertex array). These are arrays of numbers such as $\mathbf{v} \in \mathbb{R}^n$. How these numbers actually look like geometrically require another information: the basis. If we want to modify geometric realization of the object, we do not change numbers in the geometry spreadsheet. Instead, we change the basis.

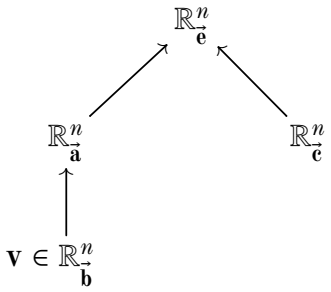
All bases are stored by matrices representing the interrelations between bases.

For example, suppose the vector depicted by $\mathbf{v}$ should be realized using the basis $\vec{\mathbf{b}}$. Suppose we have other bases $\vec{\mathbf{a}}$, $\vec{\mathbf{e}}$, $\vec{\mathbf{c}}$. Then we must provide the relationship between them with a directed tree, say



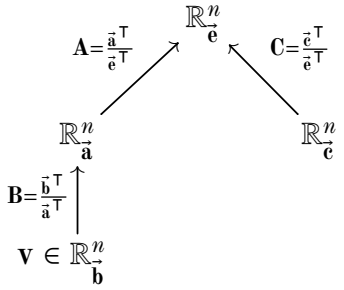
(5.33)

These bases establish a few $\mathbb{R}^n$ spaces, labeled as $\mathbb{R}^n_{\vec{\mathbf{e}}}$, $\mathbb{R}^n_{\vec{\mathbf{a}}}$, $\mathbb{R}^n_{\vec{\mathbf{b}}}$, $\mathbb{R}^n_{\vec{\mathbf{c}}}$. The labels are just to remind us that, if $\mathbf{v} \in \mathbb{R}^n_{\vec{\mathbf{b}}}$, then we should visualize it as $\vec{v} = \vec{\mathbf{b}}^\intercal \mathbf{v}$ using the indicated basis. These $\mathbb{R}^n$ spaces are connected by the same graph



(5.34)

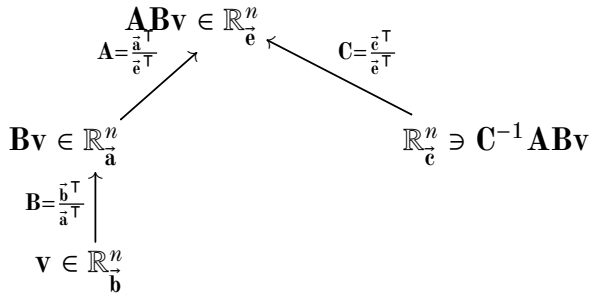
On each arrow (connection), annotate the matrix as the ratio of the source basis over the destination basis:



(5.35)

Now, for the vector $\vec{\mathbf{b}}^\intercal \mathbf{v}$ we want to represent, we can just follow the arrow and obtain all representations in every basis:

$$\vec{v} = \vec{\mathbf{b}}^\intercal(\mathbf{v}) = \vec{\mathbf{a}}^\intercal(\mathbf{Bv}) = \vec{\mathbf{e}}^\intercal(\mathbf{ABv}) = \vec{\mathbf{c}}^\intercal(\mathbf{C}^{-1}\mathbf{ABv}). \tag{5.36}$$



(5.37)

In the rendering pipeline, if the rasterization parses vectors with respect to the $\vec{\mathbf{e}}$ basis, then it is in the vertex shader that we apply the matrix $\mathbf{C}^{-1}\mathbf{AB}$ (as a uniform variable) to the vertex attribute $\mathbf{v}$. That is, the transformation is computed at render time in parallel in the GPU.

5.5 Inner product

So far, the geometry of a real vector space does not involve any measurement of length or angle. (The only geometric structure we use in Example 5.2 the vector space is parallelism.)

A real vector space with an additional structure of inner product is an **inner product space**.

Definition 5.7 Let V be a real vector space. An operator $\cdot : V \times V \to \mathbb{R}$ is an **inner product** if it is

- symmetric: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$;
- bilinear: $(\vec{u}_1 + \vec{u}_2) \cdot \vec{v} = \vec{u}_1 \cdot \vec{v} + \vec{u}_2 \cdot \vec{v}$, and $(c\vec{u}) \cdot \vec{v} = c(\vec{u} \cdot \vec{v})$; similarly $\vec{u} \cdot (\vec{v}_1 + \vec{v}_2) = \vec{u} \cdot \vec{v}_1 + \vec{u} \cdot \vec{v}_2$ and $\vec{u} \cdot (c\vec{v}) = c(\vec{u} \cdot \vec{v})$;
- positive-definite: $\vec{u} \cdot \vec{u} > 0$ for all $\vec{u} \neq \vec{0}$.

5.5.1 Lengths and angles

If our geometric vector space has a notion of angles and lengths, then we can measure the length $|\vec{u}|$ of a vector $\vec{u}$, and measure the angle θ between two vectors $\vec{u}, \vec{v}$. Such a geometric space is called a **Euclidean vector space**. (Euclidean space does not have a distinguished origin, so we call it a Euclidean "vector" space.) In a Euclidean vector space, we take

$$\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}| \cos\theta. \tag{5.38}$$

With some trigonometry, one can check that all the conditions for Definition 5.7 are fulfilled.

Conversely, once we have an inner product, we can define the notion of length and angle in an abstract real vector space.

Definition 5.8 Let V be a real vector space equipped with an inner product.

- The **length**, or the **norm**, of $\vec{v}$ is given by $|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$.
- A vector $\vec{v}$ is **unit** if $|\vec{v}| = 1$.
- **Normalization** with respect to the norm is to map $\vec{v}$ to the unit vector $\vec{v}/|\vec{v}|$.
- The angle $0 \leq \theta \leq \pi$ between two nonzero vectors $\vec{u}, \vec{v} \in V$ is given by

$$\cos(\theta) = \frac{\vec{u}}{|\vec{u}|} \cdot \frac{\vec{v}}{|\vec{v}|}. \tag{5.39}$$

That is,

$$\theta = \arccos\left(\frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}\right). \tag{5.40}$$

5.5.2 Algebraic representation

How does inner product look like under a basis? Let $\vec{\mathbf{e}}$ be a basis for V. Write $\vec{u} = \vec{\mathbf{e}}^{\intercal}\mathbf{u} = \sum_{i=1}^{n} u_i \vec{e}_i$ and $\vec{v} = \vec{\mathbf{e}}^{\intercal}\mathbf{v} = \sum_{j=1}^{n} v_j \vec{e}_j$. Then

$$\begin{aligned} \vec{u} \cdot \vec{v} &= \left(\sum_{i=1}^{n} u_i \vec{e}_i\right) \cdot \left(\sum_{j=1}^{n} v_j \vec{e}_j\right) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} u_i v_j \vec{e}_i \cdot \vec{e}_j \end{aligned} \tag{5.41}$$

In matrix notation,

$$\vec{u}\cdot\vec{v} = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix} \underbrace{\begin{bmatrix} \vec{e}_1\cdot\vec{e}_1 & \cdots & \vec{e}_1\cdot\vec{e}_n \\ \vdots & \ddots & \vdots \\ \vec{e}_n\cdot\vec{e}_1 & \cdots & \vec{e}_n\cdot\vec{e}_n \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \tag{5.42}$$

$$= \underbrace{\begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix}}_{\mathbf{u}^\mathsf{T}} \underbrace{\begin{bmatrix} \vec{e}_1 \\ \vdots \\ \vec{e}_n \end{bmatrix}}_{\vec{\mathbf{e}}} \cdot \underbrace{\begin{bmatrix} \vec{e}_1 & \cdots & \vec{e}_n \end{bmatrix}}_{\vec{\mathbf{e}}^\mathsf{T}} \underbrace{\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}}_{\mathbf{v}} \tag{5.43}$$

Summary 5.4 Let $\vec{\mathbf{e}}$ be a basis for V. Then for $\vec{u} = \vec{\mathbf{e}}^\mathsf{T}\mathbf{u} \in V$ and $\vec{v} = \vec{\mathbf{e}}^\mathsf{T}\mathbf{v} \in V$ we have

$$\vec{u}\cdot\vec{v} = (\vec{\mathbf{e}}^\mathsf{T}\mathbf{u})\cdot(\vec{\mathbf{e}}^\mathsf{T}\mathbf{v}) = \mathbf{u}^\mathsf{T}\vec{\mathbf{e}}\cdot\vec{\mathbf{e}}^\mathsf{T}\mathbf{v} = \mathbf{u}^\mathsf{T}\mathbf{M}\mathbf{v} \tag{5.44}$$

where $\mathbf{M} = \vec{\mathbf{e}}\cdot\vec{\mathbf{e}}^\mathsf{T}$. Note that $\mathbf{M}$ is a symmetric positive-definite matrix.

5.5.3 Orthonormal basis

Definition 5.9 A basis $\vec{\mathbf{e}}$ for an inner product space is **orthonormal** if

$$\vec{e}_i\cdot\vec{e}_j = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases} \tag{5.45}$$

That is, $\vec{\mathbf{e}}\cdot\vec{\mathbf{e}}^\mathsf{T} = \mathbf{I}$.

Under an orthonormal basis, we have

$$\vec{u}\cdot\vec{v} = \mathbf{u}^\mathsf{T}\mathbf{v} = \sum_{i=1}^{n} u_i v_i. \tag{5.46}$$

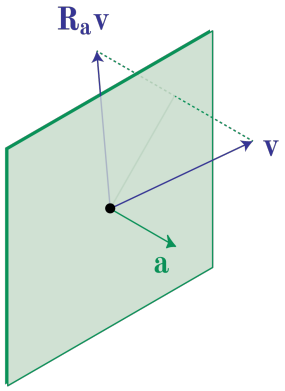
5.6 Special transformations

We review a few transformations in 2D and 3D. Throughout this section, we assume that $\vec{\mathbf{e}}$ is an orthonormal basis.

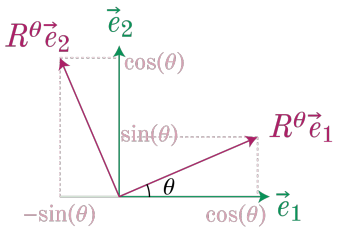
5.6.1 Rotations and reflections

Rotations and reflections are **linear isometries**. That is, they do not change the length and angles of vectors. Conversely, any linear transformation that is an isometry must be either a rotation or a reflection.

Let $R\colon V \to V$ be a rotation or a reflection. Under an orthonormal basis $\vec{\mathbf{e}}$ we have the matrix representation $R = \vec{\mathbf{e}}^\mathsf{T}\mathbf{R}$. The condition for isometry is that $(R\vec{u})\cdot(R\vec{v}) = \vec{u}\cdot\vec{v}$ for all $\vec{u},\vec{v} \in V$. So, $(\mathbf{R}\mathbf{u})^\mathsf{T}(\mathbf{R}\mathbf{v}) = \mathbf{u}^\mathsf{T}\mathbf{v}$ for all $\mathbf{u},\mathbf{v} \in \mathbb{R}^n$. This implies that $\mathbf{R}^\mathsf{T}\mathbf{R} = \mathbf{I}$.



(a) Householder transformation, *i.e.* mirror reflection.



(b) 2D rotation.

Figure 5.3 Reflection and rotation.

Summary 5.5 A linear transformation R is an isometry (rotation or reflection) if and only if its matrix representation $\mathbf{R}$ under an orthonormal basis satisfies $\mathbf{R}^\intercal\mathbf{R} = \mathbf{I}$ (equivalently, $\mathbf{R}^{-1} = \mathbf{R}^\intercal$).

Reflection with respect to a hyperplane

If $\mathbf{a}$ is a *unit* vector perpendicular to a hyperplane[1] through the origin. Then the **reflection** in the plane is given by

$$\mathbf{R_a} = \mathbf{I} - 2\mathbf{a}\mathbf{a}^\intercal. \tag{5.47}$$

A reflection in a plane is also called a **Householder transformation**.

2D rotation

The matrix that represents the counterclockwise rotation in the plane by angle θ is given by the 2-by-2 matrix:

$$\mathbf{R}^\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{5.48}$$

3D rotation

We will dive into the details of 3D rotations in the next chapter.

5.6.2 Stretching and shearing

The remaining frequently seen linear transformations are the non-isometric ones. We assume that the basis is orthonormal.

Stretching

A uniform stretching, or isotropic stretching, is a scalar times the identity

$$\mathbf{S} = s\mathbf{I}. \tag{5.49}$$

[1] hyperplane is just a plane of dimension $(n-1)$ in an n-dimensional space. So a hyperplane in 3D is a plane, and a hyperplane in 2D is a line.

A non-uniform (anisotropic) stretching along the orthonormal basis is given by a diagonal matrix. For example, in 3D,

$$\mathbf{S} = \begin{bmatrix} s_1 & & \\ & s_2 & \\ & & s_3 \end{bmatrix}. \tag{5.50}$$

For non-uniform stretching along a different set of orthonormal directions, one can just compose a rotation with a stretching. In practice, compositions of transformations are organized with a graph of bases (5.37).

Note 5.6 *A* **singular value decomposition** *(SVD) is to decompose any matrix* $\mathbf{A}$ *into* $\mathbf{A} = \mathbf{R}_1\mathbf{S}\mathbf{R}_2^\intercal$ *where both* $\mathbf{R}_1, \mathbf{R}_2^\intercal$ *are isometries, and* $\mathbf{S}$ *is a diagonal matrix with nonnegative entries. Compositions of anisotropic scaling and rotations (possibly reflections) generate all possible linear transformations.*

Shearing

A matrix such as

$$\mathbf{S} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \tag{5.51}$$

shears a rectangle into a parallelogram of the same area. These operations can also be represented using compositions of rotations and stretching (by SVD), but it is perhaps less intuitive than directly writing down (5.51).

6. 3D Rotations

No one fully understands spinors. Their algebra is formally understood but their general significance is mysterious. In some sense they describe the 'square root' of geometry and, just as understanding the square root of -1 *took centuries, the same might be true of spinors.*

Sir Michael Atiyah, 2009

Rotation is an elementary concept that we experience everyday. A way to represent 3D rotation is by a sequence of 2D rotations. This is the old technique of Euler angles. However, the Euler angles do not form a proper coordinate system over the space of all 3D rotations. Sometimes the representation is redundant, and sometimes it looses degrees of freedom. This causes many practical problems when using Euler angles.

A full understanding of the linear algebra of rotations in dimension 3 and higher is not established until the later half of 19th century. This topic in linear algebra is called **geometric algebra**. In fact, much of the linear algebra that we see today is evolved from geometric algebra. However, in the beginning of the 20th century, many geometric algebra aspects of linear algebra are "forgotten." The remaining algebra that we see today are dot product and cross product, which are seemingly enough to represent 3D rotations (and all classical mechanical systems known at the time). In the 1920's the forgotten part of the geometric algebra was partially reintroduced (reinvented) in quantum mechanics, known as spinors. Then, for a hundred years, the geometric algebra only shows up in quantum physics and pure

mathematics. In fact, for most of our engineering systems, ranging from aerospace, robotics, to a majority of computer graphics systems, the rotation is represented by Euler angles.

In recent years, geometric algebra is coming back to our civilization, starting from computer graphics. It is apparent that using an alternative representation, namely the **quaternions** (a special case of geometric algebra), overcomes many difficulties comparing to the traditional Euler angles. Many game engines use quaternions as their primary 3D rotation representation. Many people are also developing full geometric algebra math libraries for computer graphics.

6.1 Cross product

Consider a 3D Euclidean vector space V with a right-handed orthonormal basis

$$\vec{\mathbf{e}} = \begin{bmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \end{bmatrix}. \tag{6.1}$$

Definition 6.1 The (geometric) **cross product** $\vec{u} \times \vec{v}$ between $\vec{u}, \vec{v} \in V$ is the vector

$$\vec{w} = \vec{u} \times \vec{v} \in V \tag{6.2}$$

such that

- $\vec{w}$ is orthogonal to both $\vec{u}$ and $\vec{v}$ (*i.e.* $\vec{w} \cdot \vec{u} = \vec{w} \cdot \vec{v} = 0$) in the direction so that $\vec{u}, \vec{v}, \vec{w}$ forms a right-handed basis;
- $|\vec{w}| = |\vec{u}||\vec{v}| \sin\theta$ where $0 \le \theta \le \pi$ is the angle between $\vec{u}, \vec{v}$.

One can verify that $\vec{e}_x \times \vec{e}_y = \vec{e}_z$, $\vec{e}_y \times \vec{e}_z = \vec{e}_x$, $\vec{e}_z \times \vec{e}_x = \vec{e}_y$.

Note 6.1
- *Cross product is* **skew-symmetric**. *That is,* $\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$.
- *Cross product is* **not associative**. *In general,* $\vec{a} \times (\vec{b} \times \vec{c}) \neq (\vec{a} \times \vec{b}) \times \vec{c}$.

Definition 6.2 The (algebraic) **cross product** $\mathbf{u} \times \mathbf{v}$ between two arrays of numbers $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ is given by

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix} \tag{6.3}$$

One can check that under our right-handed orthonormal basis, the geometric and algebraic cross products agree with each other:

$$\vec{u} \times \vec{v} = (\vec{\mathbf{e}}\mathbf{u}) \times (\vec{\mathbf{e}}\mathbf{v}) = \vec{\mathbf{e}}(\mathbf{u} \times \mathbf{v}). \tag{6.4}$$

6.1.1 "Vector cross" as a linear transformation

Given $\vec{u}$, the action of sending $\vec{v}$ to $\vec{u} \times \vec{v}$ is a linear transformation. We write it as

$$\begin{aligned} (\vec{u}\times)\colon V &\to V, \\ \vec{v} &\mapsto \vec{u} \times \vec{v}. \end{aligned} \tag{6.5}$$

Its matrix representation $[\mathbf{u}\times]$ under the right-handed orthonormal basis

$$(\vec{u}\times) = \vec{\mathbf{e}}[\mathbf{u}\times] \tag{6.6}$$

is given by

$$[\mathbf{u}\times] = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \tag{6.7}$$

which is a skew-symmetric matrix. One can see that $[\mathbf{u}\times]\mathbf{v} = \mathbf{u}\times\mathbf{v}$:

$$\begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}. \tag{6.8}$$

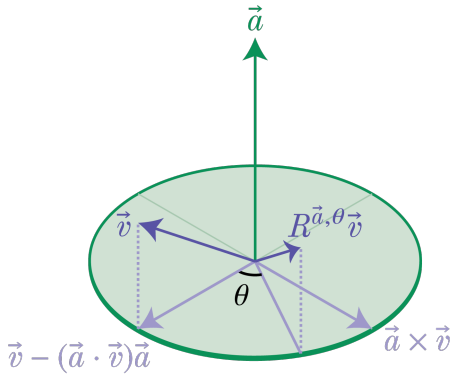
6.2 Rodrigues' axis-angle formula

Every 3D rotation can be described as a rotation about an axis.

Task 6.1 Given an axis $\vec{a}$ as a unit vector $|\vec{a}| = 1$, and given an angle θ, write down the 3D rotation $R^{\vec{a},\theta}$ as a linear operator or as a 3×3 matrix.

Suppose we apply this rotation to an arbitrary vector $\vec{v}$. We first decompose $\vec{v}$ into

$$\vec{v} = \underbrace{(\vec{a}\cdot\vec{v})\,\vec{a}}_{\substack{\text{projection of} \\ \vec{v} \text{ on the axis } \vec{a}}} + \underbrace{\vec{v} - (\vec{a}\cdot\vec{v})\,\vec{a}}_{\substack{\text{component of } \vec{v} \text{ that} \\ \text{is orthogonal to } \vec{a}}} \tag{6.9}$$



Now, observe that $\vec{a}\times\vec{v}$ is the 90° rotation of the second component $\vec{v} - (\vec{a}\cdot\vec{v})\,\vec{a}$ about $\vec{a}$. To see this, notice that $\vec{a}\times\vec{v}$ is orthogonal to both $\vec{a}$ and the $\vec{v}$, and so it is also orthogonal to their linear combination $\vec{v} - (\vec{a}\cdot\vec{v})\,\vec{a}$. Next, $\vec{a}\times\vec{v}$ and $\vec{v} - (\vec{a}\cdot\vec{v})\vec{a}$ have the same length, since $|\vec{v} - (\vec{a}\cdot\vec{v})\vec{a}|^2 = |\vec{v}|^2 - (\vec{a}\cdot\vec{v})^2 = |\vec{v}|^2(1-\cos^2\phi) = |\vec{v}|^2\sin^2\phi = |\vec{a}\times\vec{v}|^2$, where ϕ is the angle between $\vec{a}$ and $\vec{v}$. Finally, one draws a picture to convince oneself that $\vec{a}\times\vec{v}$ is the counterclockwise 90° rotation from $\vec{v} - (\vec{a}\cdot\vec{v})\vec{a}$ rather than clockwise.

Back to (6.9). Rotating the vector about $\vec{a}$ will not modify the component parallel to $\vec{a}$. It just rotates the component orthogonal to the axis as a planar rotation. So,

$$R^{\vec{a},\theta}\vec{v} = (\vec{a}\cdot\vec{v})\,\vec{a} + \cos\theta\left(\vec{v} - (\vec{a}\cdot\vec{v})\vec{a}\right) + \sin\theta\left(\vec{a}\times\vec{v}\right). \tag{6.10}$$

As an operator,

$$R^{\vec{a},\theta} = \cos\theta + (1-\cos\theta)\vec{a}(\vec{a}\cdot\,) + \sin\theta(\vec{a}\times). \tag{6.11}$$

As a matrix,

$$\mathbf{R}^{\mathbf{a},\theta} = \cos\theta\mathbf{I} + (1-\cos\theta)\mathbf{a}\mathbf{a}^{\mathsf{T}} + \sin\theta[\mathbf{a}\times] \tag{6.12}$$

Summary 6.1 Let $\mathbf{a}$ be a unit vector. Then the 3D rotation about $\mathbf{a}$ with angle θ is given by the matrix

$$\mathbf{R}^{\mathbf{a},\theta} = \cos\theta\mathbf{I} + (1-\cos\theta)\mathbf{a}\mathbf{a}^\mathsf{T} + \sin\theta[\mathbf{a}\times] \tag{6.13}$$

6.2.1 Euler–Rodrigues matrix

An alternative formula for the rotation matrix is given as follows. Let $\mathbf{a} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}$ be the unit rotation axis, and θ be the angle by which we want to rotate. Define

$$a = \cos\left(\frac{\theta}{2}\right), \quad b = \sin\left(\frac{\theta}{2}\right)a_x, \quad c = \sin\left(\frac{\theta}{2}\right)a_y, \quad d = \sin\left(\frac{\theta}{2}\right)a_z. \tag{6.14}$$

In particular $a^2+b^2+c^2+d^2 = 1$, that is, (a,b,c,d) is a unit 4D vector. Then

$$\mathbf{R}^{\mathbf{a},\theta} = \begin{bmatrix} a^2+b^2-c^2-d^2 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & a^2-b^2+c^2-d^2 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & a^2-b^2-c^2+d^2 \end{bmatrix}. \tag{6.15}$$

We will later understand (a,b,c,d) as a *quaternion*.

6.3 Euler angles

The precursor of Rodrigues' rotation (1840) is the Euler angles representation (1751). There is no real reason to replace Rodrigues' straightforward formula by Euler angles other than Euler angles' popularity and that they may be natural in some mechanical devices such as gimbal systems.

Note that when the rotation axis is aligned with a basis vector, the rotation matrix (6.13) looks like a 2D rotation matrix embedded in the 3×3 matrix

$$\mathbf{R}^{\begin{bmatrix}1\\0\\0\end{bmatrix},\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad \mathbf{R}^{\begin{bmatrix}0\\1\\0\end{bmatrix},\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \quad \mathbf{R}^{\begin{bmatrix}0\\0\\1\end{bmatrix},\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.16}$$

Then, an arbitrary 3D rotation may be represented by a product of the three elementary rotations

$$\mathbf{R} = \mathbf{R}^{\begin{bmatrix}1\\0\\0\end{bmatrix},\alpha}\mathbf{R}^{\begin{bmatrix}0\\1\\0\end{bmatrix},\beta}\mathbf{R}^{\begin{bmatrix}0\\0\\1\end{bmatrix},\gamma} \tag{6.17}$$

The angles α,β,γ are the Euler angles. There are $3! = 6$ different versions of the composition depending on the order of compositions.

The composition of rotation matrices in the Euler angle representation can be thought of as a gimbal system (Figure 6.1), in which case there is a natural hierarchy of bases, one for each ring in the gimbal system.

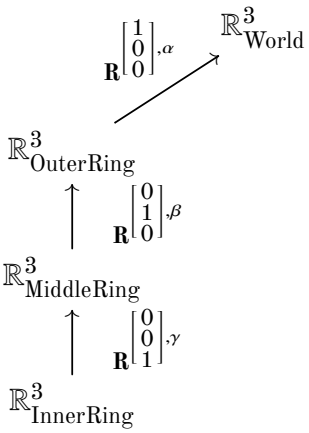


Figure 6.1 A gimbal system is a sequence of rigid rings that are hinged together consecutively along independent axes (left). The hinge angles are the Euler angles controlling the 3D rotation of the inner-most object. The hierarchy of objects are represented as a graph of bases (right).

6.3.1 Unpredictable interpolation

Euler angles have several problems. One of them is that the representation is not unique. The same 3D orientation can have infinitely many sets of Euler angles that represent it. So, from the look of the 3D orientation, we cannot infer from it what the underlying Euler angle parameters. Euler angles are additional hidden information that is outside of the visible geometry. It causes unexpected behavior when we interpolate two 3D orientations since the interpolation depends on the choice of the hidden information.

6.3.2 Gimbal lock

Another problem of Euler angles is the problem of **gimbal lock**. Even if the 3 axes $\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}$ of rotations in (6.16) seem to be independent, in reality they may not be. Note that these 3 axes, as arrays of numbers, are representing vectors in different bases, one for the outer ring of the gimbal, one for the middle ring, and one for the inner ring. So, if one of the rotating shaft (rotation axis) (most outer one) is actually aligned with another rotating shaft (most inner one) in the physical world, then the 3 degrees of freedom of rotation drops down to 2. In that case, the gimbal stalls. One usually has to manually reset the gimbal configuration when gimbal locks happen.

A gimbal lock incident happened in the Apollo 11 Moon mission. The gimbals were used for an inertial measurement unit. The gimbals were too close to a gimbal lock configuration and froze the unit. The spacecraft had to be manually moved away from the gimbal lock position in order to reset the gimbals.

6.3.3 Euler angles for cameras

Euler angles may have an advantage when we deliberately want to remove one degree of freedom in the rotation. For example, when controlling the 3D orientation of the

camera, we may want to have the up vector camera (y axis in the camera coordinate) to always point roughly up (y axis in the world coordinate). In that case, we will set the rotation axis of the outer-most ring of the gimbal to be the y axis, and we will let the camera be pointing at the rotation axis of the inner-most ring of the gimbal. As long as we freeze the inner-most axis, the up vector of the camera will not tilt sideway from the gravity direction.

6.4 Geometric algebra

So far, we have seen two different products for 3D vectors, the dot product and the cross product. It turns out that these notations were introduced in the late 19th century by Heaviside and Gibbs (1881), and dominated the linear algebra education in high schools and universities until today. So, what was there before dot and cross products?

Before Heaviside and Gibbs, we have many objects such as (6.14). This 4D vector has some pattern. The a component is a scalar, and the (b, c, d) together is parallel to the rotation axis. In fact, (a, b, c, d) together is "a scalar plus a vector," as an element of $\mathbb{R}^4 = \mathbb{R} \oplus \mathbb{R}^3$. Here $\oplus$ is called a direct sum, which is the same thing as just appending two arrays of numbers together. Heaviside and Gibbs thought that scalar and vector parts of a 4D vector are too cumbersome to parse, and therefore separate these components. As a consequence, some products get separated into a scalar product, the dot product; and a vector product, the cross product.

The problem, however, with the Heaviside–Gibbs algebra is that it *only makes sense in 3D*. Although our physical space is 3D, there have been many problems in computer science and modern physics requiring higher dimensional thinking. The Heaviside–Gibbs algebra does give us an intuition of working with general dimensions.

Another problem is that the cross product is *not associative*. It is an awkward operator in a formula, compared to addition and standard multiplications. Even though we probably have seen cross product used in many important physical laws (electromagnetism, torque, angular momentum, rotation), surprisingly, they are all replaceable by other products if we embrace the precursor of Heaviside–Gibbs algebra.

6.4.1 Insisting on associativity

Let us start with $V \cong \mathbb{R}^3$, our 3D Euclidean vector space. Between vectors we can add. In order to turn the space into an algebra, we want to be able to multiply vectors.

Definition 6.3 A **geometric product** has two requirements:

- A vector multiplied with itself gives us the length squared of the vector: $\vec{v}\vec{v} = |\vec{v}|^2 \in \mathbb{R}$.
- The multiplication is associative, that is $(\vec{x}\vec{y})\vec{z} = \vec{x}(\vec{y}\vec{z})$.

The first requirement already suggests that we should extend the space to $\mathbb{R} \oplus V$, the space of a scalar appended with a vector.

Now, consider a generic $\vec{u}+\vec{v}$ and square it: On one hand,

$$(\vec{u}+\vec{v})(\vec{u}+\vec{v}) = |\vec{u}+\vec{v}|^2 = |\vec{u}|^2 + |\vec{v}|^2 + 2\vec{u}\cdot\vec{v}. \tag{6.18}$$

On the other hand,

$$(\vec{u}+\vec{v})(\vec{u}+\vec{v}) = \vec{u}\vec{u} + \vec{u}\vec{v} + \vec{v}\vec{u} + \vec{v}\vec{v} = |\vec{u}|^2 + |\vec{v}|^2 + \vec{u}\vec{v} + \vec{v}\vec{u}. \tag{6.19}$$

Therefore,

$$\vec{u}\cdot\vec{v} = \frac{1}{2}\left(\vec{u}\vec{v} + \vec{v}\vec{u}\right). \tag{6.20}$$

Summary 6.2 Given $\vec{u},\vec{v}\in V$, as a general procedure a product $\vec{u}\vec{v}$ can be written in a symmetric part and a anti-symmetric part

$$\vec{u}\vec{v} = \frac{1}{2}\left(\vec{u}\vec{v} + \vec{v}\vec{u}\right) + \frac{1}{2}\left(\vec{u}\vec{v} - \vec{v}\vec{u}\right). \tag{6.21}$$

For a geometric product, the symmetric part must be a scalar, and it is the inner product between $\vec{u},\vec{v}$:

$$\vec{u}\vec{v} = \vec{u}\cdot\vec{v} + \frac{1}{2}\left(\vec{u}\vec{v} - \vec{v}\vec{u}\right). \tag{6.22}$$

Definition 6.4 Given $\vec{u},\vec{v}\in V$, we call

$$\vec{u}\wedge\vec{v} = \frac{1}{2}\left(\vec{u}\vec{v} - \vec{v}\vec{u}\right) \tag{6.23}$$

the **outer product** or **exterior product** between $\vec{u},\vec{v}$. Note that

$$\vec{u}\wedge\vec{v} = -\vec{v}\wedge\vec{u}. \tag{6.24}$$

An inner product takes vectors down to scalars. In contrast, the outer product takes vectors to a higher degree object (hence the name inner and outer).

At this point, we will switch to just looking at our orthonormal basis $(\vec{e}_x,\vec{e}_y,\vec{e}_z)$ for simplicity. Since $\vec{e}_x,\vec{e}_y$ have vanishing inner product, we have $\vec{e}_x\vec{e}_y = \vec{e}_x\wedge\vec{e}_y$. Now, we investigate what exactly is $\vec{e}_x\vec{e}_y$.

Summary 6.3 What we have so far in the multiplication rule is

$$\vec{e}_x\vec{e}_x = \mathbb{1},\quad \vec{e}_y\vec{e}_y = \mathbb{1},\quad \vec{e}_z\vec{e}_z = \mathbb{1}. \tag{6.25}$$

(Here, $\mathbb{1}$ denotes the basis for the scalar space $\mathbb{R}$.) $\vec{e}_x\vec{e}_y$, $\vec{e}_y\vec{e}_z$ *etc.* are yet to be investigated, but we do know that they are exterior products and therefore

$$\vec{e}_x\vec{e}_y = -\vec{e}_y\vec{e}_x,\quad \vec{e}_y\vec{e}_z = -\vec{e}_z\vec{e}_y,\quad \vec{e}_z\vec{e}_x = -\vec{e}_x\vec{e}_z. \tag{6.26}$$

Setting "$\vec{e}_x\vec{e}_y = \vec{e}_z$," and so on, will *almost* work. But that would just be the cross product, and associativity will be broken. For example $-\vec{e}_x = (\vec{e}_x \times \vec{e}_y) \times \vec{e}_y \neq \vec{e}_x(\vec{e}_y\vec{e}_y) = \vec{e}_x$.

Instead of seeking another vector in $\mathbb{R} \oplus V$ to assign to $\vec{e}_x\vec{e}_y$, we just don't. We simply extend our space $\mathbb{R} \oplus V$ to a bigger space $\mathbb{R} \oplus V \oplus \wedge^2(V)$.

Definition 6.5 — Bivectors. The space $\wedge^2(V)$ is the 3-dimensional real vector space with basis

$$\vec{\vec{e}}_{yz} \coloneqq \vec{e}_y\vec{e}_z, \quad \vec{\vec{e}}_{zx} = \vec{e}_z\vec{e}_x, \quad \vec{\vec{e}}_{xy} = \vec{e}_x\vec{e}_y. \tag{6.27}$$

That is, a generic element takes the form

$$\vec{\vec{v}} = \begin{bmatrix} \vec{\vec{e}}_{yz} & \vec{\vec{e}}_{zx} & \vec{\vec{e}}_{xy} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \vec{\vec{\mathbf{e}}}\mathbf{v} \in \wedge^2(V), \quad \mathbf{v} \in \mathbb{R}^3. \tag{6.28}$$

These elements are called **bivectors**. For later purpose, we will also use the notation $\mathbb{i}, \mathbb{j}, \mathbb{k}$ defined by

$$\mathbb{i} = -\vec{e}_y\vec{e}_z, \quad \mathbb{j} = -\vec{e}_z\vec{e}_x, \quad \mathbb{k} = -\vec{e}_x\vec{e}_y. \tag{6.29}$$

$$\mathbb{v} = \begin{bmatrix} \mathbb{i} & \mathbb{j} & \mathbb{k} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = -\vec{\vec{\mathbf{e}}}\mathbf{v} = -\vec{\vec{v}} \in \wedge^2(V). \tag{6.30}$$

Similarly:

Definition 6.6 The space $\wedge^3(V)$ is the 1-dimension real vector space with basis

$$\mathbb{I} = \vec{e}_x\vec{e}_y\vec{e}_z. \tag{6.31}$$

That is, a generic element in the space takes the form

$$p\mathbb{I} \in \wedge^3(V), \quad q \in \mathbb{R}. \tag{6.32}$$

These elements are called **pseudoscalars**.

Since we are in a 3D space, the space extension stops here. Any other product of any number of $\vec{e}_x, \vec{e}_y, \vec{e}_z$ can be reduced to one of $\mathbb{1}, \vec{e}_x, \vec{e}_y, \vec{e}_z, \mathbb{i}, \mathbb{j}, \mathbb{k}, \mathbb{I}$. For example,

$$\mathbb{i}\mathbb{I} = (-\vec{e}_y\vec{e}_z)(\vec{e}_x \underbrace{\vec{e}_y\vec{e}_z}_{\text{swap}}) = \vec{e}_y\vec{e}_z \underbrace{\vec{e}_x\vec{e}_z}_{\text{swap}}\vec{e}_y = -\vec{e}_y \underbrace{\vec{e}_z\vec{e}_z}_{\mathbb{1}}\vec{e}_x\vec{e}_y = -\vec{e}_y\vec{e}_x\vec{e}_y = \vec{e}_x\vec{e}_y\vec{e}_y = \vec{e}_x. \tag{6.33}$$

Summary 6.4 The **geometric algebra**, also known as the **Clifford algebra**, generated from V (with inner product $\cdot$) is given by the 8-dimensional space

$$\mathrm{Cl}(V) = \underbrace{\mathbb{R}}_{\text{scalars}} \oplus \underbrace{V}_{\text{vectors}} \oplus \underbrace{\wedge^2(V)}_{\text{bivectors}} \oplus \underbrace{\wedge^3(V)}_{\text{pseudoscalars}} \tag{6.34}$$

with basis given by

$$\mathbb{1}, \quad \vec{e}_x, \quad \vec{e}_y, \quad \vec{e}_z, \quad \mathbb{i} = -\vec{e}_y\vec{e}_z, \quad \mathbb{j} = -\vec{e}_z\vec{e}_x, \quad \mathbb{k} = -\vec{e}_x\vec{e}_y, \quad \mathbb{I} = \vec{e}_x\vec{e}_y\vec{e}_z. \tag{6.35}$$

The multiplication table of ab, with rows being a and columns being b, is given by

	$\mathbb{1}$	$\vec{e}_x$	$\vec{e}_y$	$\vec{e}_z$	$\mathbb{i}$	$\mathbb{j}$	$\mathbb{k}$	$\mathbb{I}$
$\mathbb{1}$	$\mathbb{1}$	$\vec{e}_x$	$\vec{e}_y$	$\vec{e}_z$	$\mathbb{i}$	$\mathbb{j}$	$\mathbb{k}$	$\mathbb{I}$
$\vec{e}_x$	$\vec{e}_x$	$\mathbb{1}$	$-\mathbb{k}$	$\mathbb{j}$	$-\mathbb{I}$	$\vec{e}_z$	$-\vec{e}_y$	$-\mathbb{i}$
$\vec{e}_y$	$\vec{e}_y$	$\mathbb{k}$	$\mathbb{1}$	$-\mathbb{i}$	$-\vec{e}_z$	$-\mathbb{I}$	$\vec{e}_x$	$-\mathbb{j}$
$\vec{e}_z$	$\vec{e}_z$	$-\mathbb{j}$	$\mathbb{i}$	$\mathbb{1}$	$\vec{e}_y$	$-\vec{e}_x$	$-\mathbb{I}$	$-\mathbb{k}$
$\mathbb{i}$	$\mathbb{i}$	$-\mathbb{I}$	$\vec{e}_z$	$-\vec{e}_y$	$-\mathbb{1}$	$\mathbb{k}$	$-\mathbb{j}$	$\vec{e}_x$
$\mathbb{j}$	$\mathbb{j}$	$-\vec{e}_z$	$-\mathbb{I}$	$\vec{e}_x$	$-\mathbb{k}$	$-\mathbb{1}$	$\mathbb{i}$	$\vec{e}_y$
$\mathbb{k}$	$\mathbb{k}$	$\vec{e}_y$	$-\vec{e}_x$	$-\mathbb{I}$	$\mathbb{j}$	$-\mathbb{i}$	$-\mathbb{1}$	$\vec{e}_z$
$\mathbb{I}$	$\mathbb{I}$	$-\mathbb{i}$	$-\mathbb{j}$	$-\mathbb{k}$	$\vec{e}_x$	$\vec{e}_y$	$\vec{e}_z$	$-\mathbb{1}$

Table 6.1 Multiplication table for the basis of the Clifford algebra $\mathrm{Cl}(V)$.

Now, we can consider the more general vectors. Similar to $\vec{v} = \vec{\mathbf{e}}\mathbf{v}$, let us write

$$\vec{\vec{v}} = \vec{\vec{\mathbf{e}}}\mathbf{v} = \underbrace{\begin{bmatrix} -\mathbb{i} & -\mathbb{j} & -\mathbb{k} \end{bmatrix}}_{\vec{\vec{\mathbf{e}}}} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{6.36}$$

for the bivector counterpart. A generic element in $\mathrm{Cl}(V)$ is some vector such as

$$a\mathbb{1} + \vec{u} + \vec{\vec{v}} + b\mathbb{I}. \tag{6.37}$$

Summary 6.5 By looking up Table 6.1, we have:

- $\vec{u}\vec{v} = (\mathbf{u}^\intercal\mathbf{v})\mathbb{1} + \vec{\mathbf{e}}(\mathbf{u}\times\mathbf{v})$.
- $\vec{u}\vec{\vec{v}} = -\vec{\mathbf{e}}(\mathbf{u}\times\mathbf{v}) + (\mathbf{u}^\intercal\mathbf{v})\mathbb{I}$.
- $\vec{\vec{u}}\vec{v} = -\vec{\mathbf{e}}(\mathbf{u}\times\mathbf{v}) + (\mathbf{u}^\intercal\mathbf{v})\mathbb{I}$.
- $\vec{\vec{u}}\vec{\vec{v}} = -(\mathbf{u}^\intercal\mathbf{v})\mathbb{1} - \vec{\vec{\mathbf{e}}}(\mathbf{u}\times\mathbf{v})$.
- $\vec{u}\mathbb{I} = \vec{\vec{u}}$.
- $\mathbb{I}\vec{u} = \vec{\vec{u}}$.
- $\vec{\vec{u}}\mathbb{I} = -\vec{u}$.
- $\mathbb{I}\vec{\vec{u}} = -\vec{u}$.

The entire algebra arises naturally by insisting on associativity. In fact, the algebra is uniquely determined by the inner product, which is the starting point.

The cross product shows up automatically, but with a more delicate relationship between vectors and bivectors. The construction works in any dimension for any inner products.[1]

6.4.2 Geometric interpretation

There are many interpretations of objects such as vectors and bivectors. The most common interpretation is that the vectors are arrows, and bivectors are the "area vector" whose magnitude is the area and the direction is the normal of the plane.

However, here we take another interpretation.

A vector $\vec{v}$ represents a weighted plane (passing through the origin) whose normal is $\vec{v}$. The weight is its magnitude $|\vec{v}|$. The exterior product $\vec{u} \wedge \vec{v}$ is a bivector, which represents an axis or an arrow. This axis is the meeting line of the two planes $\vec{u}, \vec{v}$. If $\vec{u}, \vec{v}$ have unit norm, then $\vec{u} \wedge \vec{v}$ has a size of $\sin(\theta)$ where θ is the angle between the plane.

Summary 6.6 The geometric interpretation (in our version of geometric algebra) is that $\vec{v}$ is a weighted oriented plane and $\vec{\vec{v}}$ is an arrow or a weighted oriented line. These planes and lines pass through the origin.

Theorem 6.2 — Orthogonal complement. Let $\vec{u}$ be a unit plane. Then $\mathbb{I}\vec{u} = \vec{\vec{u}}$ is the unit oriented line normal to the plane. Conversely, $-\mathbb{I}\vec{\vec{u}}$ is the plane normal to the line $\vec{\vec{u}}$.

Theorem 6.3 — The meet of two planes. Suppose there are two planes $\vec{u}, \vec{v}$ with unit norm $|\vec{u}| = |\vec{v}| = 1$ meeting at a line $\vec{\vec{\ell}}$ at an angle $0 < \theta < \pi$. Here, $\vec{\vec{\ell}}$ has unit norm ($|\vec{\vec{\ell}}\vec{\ell}| = 1$), oriented in the direction of the cross product of the normals of the planes in the order $\vec{u}$, $\vec{v}$. Then the the geometric product $\vec{u}\vec{v}$ is given by[a]

$$\vec{u}\vec{v} = \underbrace{\cos(\theta)}_{\vec{u}\cdot\vec{v}} + \underbrace{\sin(\theta)\vec{\vec{\ell}}}_{\vec{u}\wedge\vec{v}} = e^{\theta\vec{\vec{\ell}}}. \tag{6.38}$$

[a] Here, the shorthand $e^{\theta\vec{\vec{\ell}}} = \cos(\theta) + \sin(\theta)\vec{\vec{\ell}}$ works for the same reason as $e^{\theta\mathbb{i}} = \cos(\theta) + \mathbb{i}\sin(\theta)$ for complex numbers, since $\vec{\vec{\ell}}^2 = -\mathbb{1}$.

Theorem 6.4 — The join of two lines. Let $\vec{\vec{u}}, \vec{\vec{v}}$ be two lines of unit norm, spanning a plane $\vec{w}$. Let θ be the angle between $\vec{\vec{u}}, \vec{\vec{v}}$. Let $\vec{w}$ be of unit norm oriented so that $\mathbf{w}$

[1] In fact, the inner product only needs to be symmetric and bilinear. The inner product does not need to be positive definite or non-degenerate.

is in the same direction as $\mathbf{u} \times \mathbf{v}$. Then[a]

$$\vec{u}\mathbb{I}\vec{v} = -\cos(\theta)\mathbb{I} + \underbrace{\sin(\theta)\vec{w}}_{\vec{u}\vee\vec{v}}. \tag{6.39}$$

[a] Here, the join operator $\vee$ is give by $A \vee B = -\mathbb{I}((\mathbb{I}A) \wedge (\mathbb{I}B))$. We won't use this operator often.

Theorem 6.5 — Mirror reflection. Let $\vec{a}$ be a plane. Then the mirror reflection of an arrow $\vec{v}$ with respect to $\vec{a}$ is given by

$$-\vec{a}^{-1}\vec{v}\vec{a}. \tag{6.40}$$

Here $\vec{a}^{-1} = \vec{a}/|\vec{a}|^2$ so $\vec{a}^{-1}\vec{a} = \mathbb{1}$.

Proof. Since the expression $-\vec{a}^{-1}\vec{v}\vec{a}$ is just $= -\frac{\vec{a}\vec{v}\vec{a}}{|\vec{a}|^2} = -\frac{\vec{a}}{|\vec{a}|}\vec{v}\frac{\vec{a}}{|\vec{a}|}$, we may assume that $\vec{a}$ is normalized and show that $-\vec{a}\vec{v}\vec{a}$ is indeed the reflection of $\vec{v}$: Using Summary 6.5

$$\begin{aligned} -\vec{a}\vec{v}\vec{a} &= -\left(-\vec{\mathbf{e}}(\mathbf{a}\times\mathbf{v}) + (\mathbf{a}^\intercal\mathbf{v})\mathbb{I}\right)\vec{a} && (6.41)\\ &= \underbrace{\mathbf{a}^\intercal(\mathbf{a}\times\mathbf{v})}_{=0}\mathbb{1} + \vec{\mathbf{e}}\underbrace{((\mathbf{a}\times\mathbf{v})\times\mathbf{a})}_{\mathbf{v}-(\mathbf{a}^\intercal\mathbf{v})\mathbf{a}} - \vec{\mathbf{e}}(\mathbf{a}^\intercal\mathbf{v})\mathbf{a} && (6.42)\\ &= \vec{\mathbf{e}}\left(\mathbf{v} - 2(\mathbf{a}^\intercal\mathbf{v})\mathbf{a}\right) && (6.43) \end{aligned}$$

which is indeed the Householder reflection. □

6.4.3 3D rotation

A 3D rotation is just a composition of two mirror reflections. Suppose there are two planar mirrors meeting along an axis at an angle ϕ. Now, stand in between the mirror and face the axis. Then the image of ourselves next to us on our right is the reflection by the mirror to our right. The second image to our right is the reflection first by the left mirror and then by the right mirror. This second image is no-longer a mirrored reflection of ourselves, but a rotated version of ourselves. It is precisely the counterclockwise rotation of ourselves about the axis by 2ϕ.

So, to construct the rotation about an axis $\vec{a}$ (normalized: $\vec{a}\vec{a} = -1$) by an angle θ, consider two arbitrary unit planes $\vec{p}$ and $\vec{q}$ meeting along $\vec{a}$ at angle $\theta/2$. By Theorem 6.3, this condition is described algebraically as

$$\vec{p}\vec{q} = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\vec{a} = e^{\frac{\theta}{2}\vec{a}}. \tag{6.44}$$

The fact that the rotation is given by the consecutive mirror reflections (Theorem 6.5)

yields the formula:

$$R^{\vec{a},\theta}\vec{v} = \vec{q}^{-1}\left(\vec{p}^{-1}\vec{v}\vec{p}\right)\vec{q} \tag{6.45}$$

$$= \vec{q}^{-1}\vec{p}^{-1}\vec{v}\vec{p}\vec{q} \tag{6.46}$$

$$= (\vec{p}\vec{q})^{-1}\vec{v}\vec{p}\vec{q} \tag{6.47}$$

$$= e^{-\frac{\theta}{2}\vec{a}}\vec{v}e^{\frac{\theta}{2}\vec{a}} \tag{6.48}$$

$$= \left(\cos\left(\tfrac{\theta}{2}\right) - \sin\left(\tfrac{\theta}{2}\right)\vec{a}\right)\vec{v}\left(\cos\left(\tfrac{\theta}{2}\right) + \sin\left(\tfrac{\theta}{2}\right)\vec{a}\right). \tag{6.49}$$

Summary 6.7 The rotation about an axis $\vec{a}$ (normalized $\mathbf{a}^\intercal\mathbf{a} = 1$) by angle θ can be achieved by a **rotor** or also called a **spinor**

$$\psi := \left(\cos\left(\tfrac{\theta}{2}\right) + \sin\left(\tfrac{\theta}{2}\right)\vec{a}\right) = e^{\frac{\theta}{2}\vec{a}} \in \mathbb{R}\oplus\wedge^2(V). \tag{6.50}$$

Note that the multiplicative inverse of ψ is given by

$$\psi^{-1} = \psi := \left(\cos\left(\tfrac{\theta}{2}\right) - \sin\left(\tfrac{\theta}{2}\right)\vec{a}\right) = e^{-\frac{\theta}{2}\vec{a}} \in \mathbb{R}\oplus\wedge^2(V). \tag{6.51}$$

Then, the desired rotation is given by

$$R^{\vec{a},\theta}\vec{v} = \psi^{-1}\vec{v}\psi. \tag{6.52}$$

This construction of rotation works in any dimensions and for any inner product structure. The spinor is an (invertible) element in the even-subalgebra $\mathbb{R}\oplus\wedge^2(V)\oplus\wedge^4(V)\oplus\cdots\oplus\wedge^{2\lfloor n/2\rfloor}(V)$. For example, the Dirac spinors in quantum mechanics arise as the spinor for the four dimensional spacetime with Minkowski metric in special relativity.

Note 6.2 *It is worth noting that there are two ψ's representing one rotation R. That is why it is often said that spinors are square-roots of rotations. In fact, for each rotation, there are exactly two normalized spinors (rotors) ψ and $-\psi$ that describes that same rotation.*

6.5 Quaternions

The quaternion algebra is a simplified version of the geometric algebra in 3D. Historically, quaternions came first, introduced by Hamilton in 1843. Soon after the discovery of quaternion, Clifford, Cayley, Grassmann *etc.* developed the geometric algebra as the natural geometric explanation of the quaternions. The modern vector algebra is a simplification of geometric algebra by Heaviside and Gibbs. Quaternions may look nothing like modern linear algebra, but as a matter of fact quaternion is the origin of the modern linear algebra.

Quaternions are widely used in computer graphics. Many math libraries for graphics applications have quaternions. Many game engines and commercial softwares

graphics software rotates object based on quaternions.

The original motivation for Hamilton to discover quaternions is to generalize complex numbers. Unit complex numbers such as $e^{\mathbb{i}\theta} = \cos(\theta) + \mathbb{i}\sin(\theta)$ can rotate complex numbers $a + \mathbb{i}b$ by complex algebra:

$$e^{\mathbb{i}\theta}(a + \mathbb{i}b) = (\cos(\theta)a - \sin(\theta)b) + (\sin(\theta)a + \cos(\theta)b)\mathbb{i} \tag{6.53}$$

which reproduces the 2D rotation

$$\begin{bmatrix}\cos(\theta)a - \sin(\theta)b\\ \sin(\theta)a + \cos(\theta)b\end{bmatrix} = \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix}. \tag{6.54}$$

In 1843, Hamilton found the analogue for 3D rotations. The space of quaternions is denoted by $\mathbb{H}$, which is a 4-dimensional real vector space. A quaternion is

$$q = a + b\mathbb{i} + c\mathbb{j} + d\mathbb{k} \in \mathbb{H}, \quad a, b, c, d \in \mathbb{R}. \tag{6.55}$$

The multiplication rules are precisely given in Table 6.1:

$$\mathbb{i}^2 = -\mathbb{1}, \quad \mathbb{j}^2 = -\mathbb{1}, \quad \mathbb{k}^2 = -\mathbb{1}, \tag{6.56}$$

$$\mathbb{i}\mathbb{j} = -\mathbb{j}\mathbb{i} = \mathbb{k}, \quad \mathbb{j}\mathbb{k} = -\mathbb{k}\mathbb{j} = \mathbb{i}, \quad \mathbb{k}\mathbb{i} = -\mathbb{i}\mathbb{k} = \mathbb{j}. \tag{6.57}$$

In other words, quaternions are scalars and bivectors in the geometric algebra built from 3D. The quaternion multiplication is associative, as a property inherited from the geometric algebra. Do note that there are minus signs from the standard bases of bivectors

$$\mathbb{i} = -\vec{e}_{yz}, \quad \mathbb{j} = -\vec{e}_{zx}, \quad \mathbb{k} = -\vec{e}_{xy}. \tag{6.58}$$

(It will still work without the minus signs, but then we will have $\mathbb{i}\mathbb{j} = -\mathbb{k}$ instead of $\mathbb{i}\mathbb{j} = \mathbb{k}$.) In particular, for imaginary quaternions $\mathbb{u} = u_x\mathbb{i} + u_y\mathbb{j} + u_z\mathbb{k}$, $\mathbb{v} = v_x\mathbb{i} + v_y\mathbb{j} + v_z\mathbb{k}$

$$\mathbb{u}\mathbb{v} = -(\mathbf{u}^\intercal\mathbf{v}) + \begin{bmatrix}\mathbb{i} & \mathbb{j} & \mathbb{k}\end{bmatrix}(\mathbf{u}\times\mathbf{v}). \tag{6.59}$$

Since quaternions are thought of as generalized complex numbers, there are some complex number terminologies in the context of quaternions. Each quaternion has a real part and an imaginary part

$$\underbrace{a}_{\text{real}} + \underbrace{b\mathbb{i} + c\mathbb{j} + d\mathbb{k}}_{\text{imaginary}}. \tag{6.60}$$

A purely imaginary quaternion is a 3D vector

$$\mathbb{v} = v_x\mathbb{i} + v_y\mathbb{j} + v_z\mathbb{k} \in \mathrm{Im}(\mathbb{H}) \cong \mathbb{R}^3 \tag{6.61}$$

or more precisely a bivector (with an extra minus sign) $\mathbb{v} = -\vec{v} = \vec{\mathbf{e}}\mathbf{v}$. The quaternion conjugation is similar to the complex conjugation:

$$\overline{(a + b\mathbb{i} + c\mathbb{j} + d\mathbb{k})} = a - b\mathbb{i} - c\mathbb{j} - d\mathbb{k}. \tag{6.62}$$

Note that conjugation reverses the order: for two quaternions $p, q \in \mathbb{H}$,

$$\overline{pq} = \bar{q}\,\bar{p}. \tag{6.63}$$

The squared norm of a quaternion is

$$|a + b\mathbb{i} + c\mathbb{j} + d\mathbb{k}|^2 = \overline{(a + b\mathbb{i} + c\mathbb{j} + d\mathbb{k})}(a + b\mathbb{i} + c\mathbb{j} + d\mathbb{k}) = a^2 + b^2 + c^2 + d^2. \tag{6.64}$$

The norm can distribute over the products

$$|pq| = |p||q|. \tag{6.65}$$

The multiplicative inverse of a quaternion is given by

$$q^{-1} = \frac{\bar{q}}{|q|^2}. \tag{6.66}$$

3D vectors can be rotated by quaternions in the same as Summary 6.7. The only difference is that there is an extra minus sign when we translate bivectors to imaginary quaternions.

Summary 6.8 The rotation about an axis $\mathbf{a} \in \mathbb{R}^3$ (normalized $\mathbf{a}^\intercal\mathbf{a} = 1$) by angle θ can be achieved by quaternions. Construct $\mathbb{a} = a_x\mathbb{i} + a_y\mathbb{j} + a_z\mathbb{k}$, and consider

$$q \coloneqq \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\mathbb{a} = e^{\frac{\theta}{2}\mathbb{a}} \in \mathbb{H} \tag{6.67}$$

which is $q = \bar{\psi}$ since $\mathbb{a} = -\vec{a}$ compared to (6.50). Each 3D vector $\mathbf{v} \in \mathbb{R}^3$ can be thought of as an imaginary quaternion $\mathbb{v} = v_x\mathbb{i} + v_y\mathbb{j} + v_z\mathbb{k}$. Then, the desired rotation is given by

$$R^{\mathbb{a},\theta}\mathbb{v} = q\mathbb{v}q^{-1}. \tag{6.68}$$

6.6 Exercise

Exercise 6.1 Write a function that computes the 3×3 rotation matrix given the axis of rotation and the rotation angle.

```
// the constant M_PI requires #include<math.h>
glm::mat3 rotation(const float degrees, const glm::vec3 axis){
    const float angle = degrees * M_PI/180.0f; // convert to radians
    const glm::vec3 a = glm::normalize(axis);
    glm::mat3 R;

    ...

    return R;
}
```

This function will be useful for the exercise of the next chapter. ▪

In, GLM ans GLSL, the matrix indices are ordered by the *column-major* ordering. That is, for a `glm::mat3 M`, M_{ij} corresponds to `M[j][i]`. In fact, `M[j]` gives us a `glm::vec3` object, corresponding to the j-th column of M.

7. Affine Geometry

To visualize a geometric object in computer graphics, we want to prescribe the **position** of a point or a vertex. The notion of position is the focus of this chapter.

7.1 Positions and displacements

Positions are similar to vectors. Under a coordinate system, a position can be described by an array of numbers, just like how vectors are an array of numbers under a basis. You may have even heard of the term "position vector." However, as a common practice in computer graphics, we distinguish the notion of positions and vectors.

The mathematical notion of vectors depicts the physical notion of **displacement**, typically visualized as an **arrow**. Vectors are elements in a vector space. We can add two arbitrary vectors (consecutive displacement). We can scale a vector (resize the displacement). The zero vector, which is built-in in any vector space, is a meaningful notion when we think of "no displacement."

In contrast, a **position** is to describe the location of a **point**, instead of an arrow. There is no obvious way we can add two points and get another point. We do not scale point. A "zero" element of position is also unnatural. One may argue that "the origin is the zero." But in physical reality, there is no origin. The choice of origin is artificial. One may also argue that one can just add or scale positions by drawing an arrow from the origin, but then it depends on the choice of origin, and in that case we are just talking about positions as displacements from the origin rather than positions in their own right.

What we could do to positions is to **translate** or **displace** a set of points. On the other hand, it is less natural to do so in the space of displacements (vectors). For example, adding a constant vector will translate zero vector to some other vector, and thus not a linear transformation.

Definition 7.1 A vector is decorated with an arrow: $\vec{v}$. A position is denoted with an underbar: $\underline{p}$. Boldface symbol $\mathbf{u}$ denotes an array of numbers.

Based on our physical/geometric intuition,

1. We can take the difference of two positions and get a displacement: $\underline{q} - \underline{p} = \vec{v}$.
2. (Equivalent to above.) A position can be added by a displacement to yield another position: $\underline{p} + \vec{v} = \underline{q}$.
3. We can linear combine displacements: $c_1\vec{u} + c_2\vec{v} = \vec{w}$.
4. We do not add positions or scale positions.
5. We are allowed to take the average of two positions: $\frac{1}{2}\underline{p} + \frac{1}{2}\underline{q} = \underline{r}$. (We will have a more general version of averaging.) But a more general linear combination between position is still not allowed.

7.1.1 An extra component as an indicator

To build these rules into arithmetic expressions, we adopt the following setup. Each 3D *position/point* with coordinate values $\mathbf{p} = (p_x, p_y, p_z)$ is expressed by the 4D array

$$\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \tag{7.1}$$

and each 3D *displacement/vector* with components $\mathbf{v} = (v_x, v_y, v_z)$ is expressed by the 4D array

$$\begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}. \tag{7.2}$$

Then, indeed,

1. The difference of two positions yield a displacement: $\begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} - \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{q}-\mathbf{p} \\ 0 \end{bmatrix}$.
2. A position plus a displacement is a position: $\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} + \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{p}+\mathbf{v} \\ 1 \end{bmatrix}$.
3. Linear combinations of displacements are displacements: $c_1 \begin{bmatrix} \mathbf{u} \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} c_1\mathbf{u}+c_2\mathbf{v} \\ 0 \end{bmatrix}$.
4. We cannot add two positions: $\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} + \begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{p}+\mathbf{q} \\ 2 \end{bmatrix}$, which is not yet declared.
5. The average of positions is allowed: $\frac{1}{2}\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(\mathbf{p}+\mathbf{q}) \\ 1 \end{bmatrix}$.

Summary 7.1 In 3D computer graphics, we introduce a 4th coordinate (the w component, after the x, y, z components) to indicate position/vector.

Definition 7.2 The 4D array representation for positions and vectors is called the **homogeneous coordinate**.

So far, the utility of the 4th coordinate seems to be just the indicator for position/vector. In the next few sections, we will see more and more utilities of this extra dimension.

7.1.2 Other kinds of vectors

At this point, we should mention that there are other kinds of vectors that are not interpreted as displacements. The classification of different kinds of vectors are sometimes not in an agreement, depending on the conventions and scientific communities.

There are keywords such as "covariant" verses "contravariant" vectors, "polar" verses "axial" vectors, pseudovectors, covectors, differential forms, bivectors, *etc.*

Arrow of displacement (vector) displacement, velocity, acceleration, tangent vector,	**Directed line density (flux)** current, electric displacement,
Plane of impulse (area vector) tangent plane, normal vector, area vector,	**Directed plane density (covector)** wave vector, momentum, force, gradient, electric field,
Plane/axis of rotation (axial vector) rotation axis, normal vector, angular velocity, angular acceleration,	**Rotation axis density (2-form)** angular momentum, torque, magnetic field, vorticity,

The reason to distinguish different kinds of vectors is that different types of vectors transform differently. For example, scaling the space larger will lengthen an arrow but decrease the gradient of a function.

Note 7.1 *In the introductory computer graphics, the only thing we should pay attention is the* **normal vector**. *The normal vector is an important information for lighting. Normal vector does transform differently if we anistropically stretch or shear the coordinates.*

7.2 Affine space

Similar to that the collection of all vectors form a vector space, the collection of all positions also have a name. The set of all positions form a so-called **affine space**.

Definition 7.3 An **affine space** P modeled on a vector space V is a set of points with an addition (or translation) operator

$$+ : V \times P \to P \tag{7.3}$$
$$(\vec{v}, \underline{p}) \mapsto \vec{v} + \underline{p} \tag{7.4}$$

such that

- $(\vec{u} + \vec{v}) + \underline{p} = (\vec{u}) + (\vec{v} + \underline{p})$ for all $\vec{p} \in P$ and $\vec{u}, \vec{v} \in V$.
- $\vec{0} + \underline{p} = \underline{p}$ for all $\vec{p} \in P$.
- For any $\underline{p}, \underline{q} \in P$, there exists one and only one $\vec{v} \in V$ such that $\underline{q} = \vec{v} + \underline{p}$. This vector $\vec{v}$ is also denoted as $\vec{v} = \underline{q} - \underline{p}$.

A position is also called an **affine point**.

This definition formally describe our observation that affine points can be added by a displacement. The definition of the affine space also suggests that, if we fix any point, called the origin $\underline{o} \in P$, then P is exactly V, as any point $\underline{p} \in P$ can be identified as the vector $\underline{p} - \underline{o}$.

Summary 7.2 An affine space is a vector space without an origin. Take an affine space and pick an origin, then we have a vector space.

▪ **Example 7.1 — Algebraic.** The positions expressed as (7.1) form an affine space. Namely, $P = \{\mathbf{r} = (r_x, r_y, r_z, r_w) \in \mathbb{R}^4 \,|\, p_w = 1\}$ is an affine space modeled on the 3-dimensional vector space $V = \{\mathbf{r} = (r_x, r_y, r_z, r_w) \,|\, r_w = 0\} = \mathbb{R}^3$. ▪

▪ **Example 7.2 — Geometric.** Similar to Example 5.2, the collection of points in a flat plane (a flat 3D space, or higher dimensional spaces) is an affine space. The only requirement is that the space has a notion of parallelism. In other words, take a geometric vector space and forget about the origin. ▪

7.2.1 Coordinate system

Recall that, for vector spaces, we can bridge the algebraic representation and the geometric picture of a vector space by a *basis*. Similarly, for affine spaces, we can bridge the algebra and geometry together by an **affine coordinate system**.

Definition 7.4 Suppose P is an affine space modeled on a vector space V. Then an **affine coordinate system** is a basis $\vec{e}_1, \dots, \vec{e}_n$ for V together with a point $\underline{o} \in P$.

Given a coordinate system as such, every point $\underline{p} \in P$ is uniquely represented by numbers $p_1, \dots, p_n \in \mathbb{R}$ as

$$\underline{p} = p_1 \vec{e}_1 + \cdots + p_n \vec{e}_n + \underline{o}. \tag{7.5}$$

For shorthand, we write

$$\underline{p} = \begin{bmatrix} \vec{e}_1 & \cdots & \vec{e}_n & \underline{o} \end{bmatrix} \begin{bmatrix} p_1 \\ \vdots \\ p_n \\ 1 \end{bmatrix}. \tag{7.6}$$

Summary 7.3 Take an affine coordinate system $[\,\vec{\mathbf{e}}\ \underline{o}\,] = [\,\vec{e}_x\ \vec{e}_y\ \vec{e}_z\ \underline{o}\,]$ for the 3D affine space P, each position $\underline{p} \in P$ has a 3D array of numbers $\mathbf{p}_{3D} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \in \mathbb{R}^3$, and $\mathbf{p}_{4D} = \begin{bmatrix} \mathbf{p}_{3D} \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$ so that

$$\underline{p} = \begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix} \mathbf{p}_{4D}. \tag{7.7}$$

7.3 Affine transformation

Under a coordinate system, each point $\underline{p}$ is given by a 3D array of numbers $\mathbf{p}_{3D}$ or a 4D array of numbers $\mathbf{p}_{4D}$ with a "dummy" last entry $p_w = 1$.

In terms of arrays of numbers, an **affine transformation** is the operation

$$\begin{array}{cc} \text{before transformation} & \text{after transformation} \\ \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \mapsto & \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} \\ a_{yx} & a_{yy} & a_{yz} \\ a_{zx} & a_{zy} & a_{zz} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \end{array} \tag{7.8}$$

$$\mathbf{p}_{3D} \mapsto \mathbf{A}_{3\times 3}\mathbf{p}_{3D} + \mathbf{b} \tag{7.9}$$

That is, affine transformations are linear transformation with possibly with an additional translation. Using the 4D homogeneous coordinate, an affine transformation can be written as a *linear* transformation (described by a single matrix-vector multiplication)

$$\begin{array}{cc} \text{before transformation} & \text{after transformation} \\ \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \mapsto & \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} & b_x \\ a_{yx} & a_{yy} & a_{yz} & b_y \\ a_{zx} & a_{zy} & a_{zz} & b_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \end{array} \tag{7.10}$$

$$\begin{bmatrix} \mathbf{p}_{3D} \\ 1 \end{bmatrix} \mapsto \underbrace{\begin{bmatrix} \mathbf{A}_{3\times 3} & \mathbf{b} \\ & 1 \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} \mathbf{p}_{3D} \\ 1 \end{bmatrix}}_{\mathbf{p}_{4D}} \tag{7.11}$$

7.3.1 Coordinate-free definition

We can also define affine transformations formally without talking about coordinates:

Definition 7.5 Let P be an affine space modeled on a vector space V. A map $M\colon P \to P$ is called an **affine transformation** if there exists a linear transformation $A\colon V \to V$ such that

$$M(\underline{q}) - M(\underline{p}) = A(\underline{q} - \underline{p}) \quad \text{for all } \underline{p}, \underline{q} \in P. \tag{7.12}$$

That is, *affine transformations are the ones that transform displacement vectors linearly.*

If we take an origin $\underline{o} \in P$ and write $\underline{p} = \underline{o} + \vec{p}$. Then we must have $M(\underline{p}) - M(\underline{o}) = A(\vec{p})$ by definition. Therefore,

$$M(\underline{p}) = A(\vec{p}) + M(\underline{o}) \tag{7.13}$$

which is the linear transformation of the original "position vector" $\vec{p}$ with an additional translation by $M(\underline{o})$.

Summary 7.4 An affine transformation M has a linear part A that linearly trans-

forms displacement vectors. If $\underline{p} = \vec{p} + \underline{o}$, then

$$M(\underline{p}) = A(\vec{p}) + M(\underline{o}). \tag{7.14}$$

7.3.2 Matrix representation of an affine transformation

Suppose we have a coordinate system on P

$$\begin{bmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z & \underline{o} \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix}. \tag{7.15}$$

Let M be an affine transformation on P with A being its linear part. Then, a point $\underline{p} \in P$ is transformed as follows:

$$\begin{aligned} M(\underline{p}) &= M\left(p_x\vec{e}_x + p_y\vec{e}_y + p_z\vec{e}_z + \underline{o}\right) && (7.16)\\ &= A(p_x\vec{e}_x + p_y\vec{e}_y + p_z\vec{e}_z) + M(\underline{o}) && (7.17)\\ &= p_x(A\vec{e}_x) + p_y(A\vec{e}_y) + p_z(A\vec{e}_z) + M(\underline{o}). && (7.18) \end{aligned}$$

That is, the transformed point $M(\underline{p})$ is described by the same coordinate values $(p_x, p_y, p_z) \in \mathbb{R}^3$ but they are realized using a new coordinate system

$$\begin{bmatrix} A\vec{e}_x & A\vec{e}_y & A\vec{e}_z & M(\underline{o}) \end{bmatrix} = \begin{bmatrix} A\vec{\mathbf{e}} & M(\underline{o}) \end{bmatrix}. \tag{7.19}$$

Now, we can re-express the new coordinate system (the new basis and the new origin) in terms of the old coordinate system. This gives us a 3-by-3 matrix $\mathbf{A}$ and $\mathbf{b} \in \mathbb{R}^3$:

$$A\vec{\mathbf{e}} = \vec{\mathbf{e}}\mathbf{A}, \quad M(\underline{o}) = \vec{\mathbf{e}}\mathbf{b} + \underline{o}. \tag{7.20}$$

Therefore,

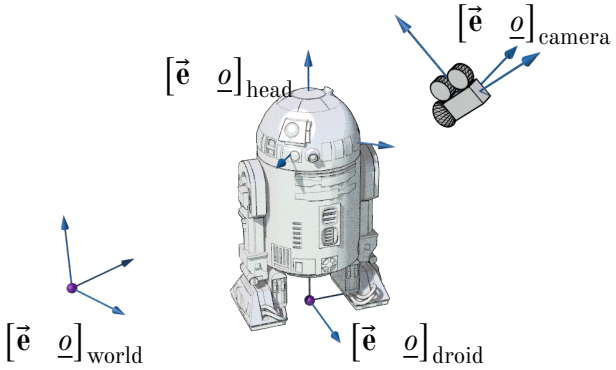
$$\begin{bmatrix} A\vec{\mathbf{e}} & M(\underline{o}) \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{e}}\mathbf{A} & \vec{\mathbf{e}}\mathbf{b} + \underline{o} \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix}. \tag{7.21}$$

Summary 7.5 Under a coordinate system $\begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix}$, an affine transformation M with linear part A has a 4-by-4 matrix representation

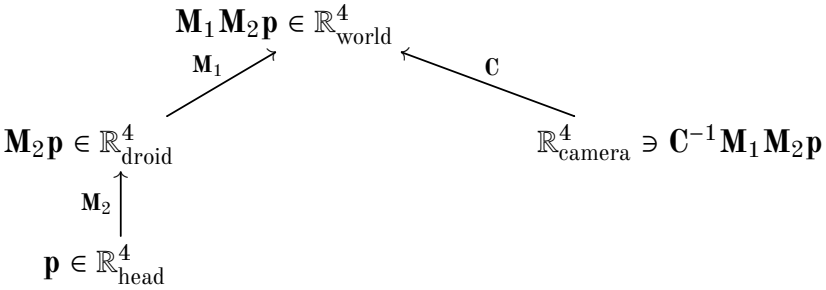
$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} \tag{7.22}$$

where $\mathbf{A}$ is the 3-by-3 matrix representation for A under the basis $\vec{\mathbf{e}}$, and $\mathbf{b} \in \mathbb{R}^3$ is some translation component. Specifically,

$$M\begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix} = \begin{bmatrix} A\vec{\mathbf{e}} & M(\underline{o}) \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{e}} & \underline{o} \end{bmatrix} \mathbf{M}. \tag{7.23}$$



(a) A scene with a complex geometry and a camera.



(b) A scene graph with transformation matrices describing the relationship between coordinate frames.

Figure 7.1 A typical scene in a graphics application.

For a general point $\underline{p} = [\vec{\mathbf{e}} \quad \underline{o}] \begin{bmatrix} \mathbf{p}_{3D} \\ 1 \end{bmatrix}$ or $\underline{p} = [\vec{\mathbf{e}} \quad \underline{o}]\, \mathbf{p}_{4D}$, we find

$$M\left([\vec{\mathbf{e}} \quad \underline{o}]\, \mathbf{p}_{4D}\right) = [\vec{\mathbf{e}} \quad \underline{o}]\, \mathbf{M}\mathbf{p}_{4D}. \tag{7.24}$$

7.4 Affine transformations in computer graphics

Now we have the same story as in Section 5.4. In computer graphics, every point position is encoded in a 4D array $\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$ with the last entry equal to 1. The 4D array of numbers alone does not tell us where the position is. It has to be aided with an affine coordinate frame.

In a scene, there can be many coordinate systems. These coordinate systems are interconnected into a tree, called a **scene graph**. If $[\vec{\mathbf{e}}_1 \quad \underline{o}_1]$, $[\vec{\mathbf{e}}_2 \quad \underline{o}_2]$ are two coordinates connected in the scene graph, then their relationship is encoded in a 4-by-4 matrix $\mathbf{M}$:

$$[\vec{\mathbf{e}}_2 \quad \underline{o}_2] = [\vec{\mathbf{e}}_1 \quad \underline{o}_1]\, \mathbf{M}. \tag{7.25}$$

A typical scene contains many geometric objects such as in Figure 7.1. Suppose we have a geometry of a bolt in the head. It is easier and more intuitive to have the bolt's vertex position be described relative to head's coordinate system. Now, we just need to know the relative transformation between the head's coordinate frame and the droid's frame. A modification of this transformation will bring along hundreds of other parts of the head together. Similarly, once the droid is grouped, we describe the droid's placement in the world by specifying the transformation matrix between the droid's frame and the world's frame. Finally, we have a camera, which also has a transformation relative to the world.

Definition 7.6 — Model matrix. The 4-by-4 transformation matrices $\mathbf{M}$'s in Figure 7.1 (pointing from the leaf of the graph back to the world frame) are called the **model matrices**. For example, in the figure, the model matrix of the head relative to the droid is $\mathbf{M}_2$. The model matrix of the head relative to the world is $\mathbf{M}_1\mathbf{M}_2$.

Definition 7.7 — Camera matrix. The **camera matrix** is the model matrix $\mathbf{C}$ of the camera relative to the world.

Definition 7.8 — View matrix. The **view matrix** $\mathbf{V}$ is the inverse of the camera matrix

$$\mathbf{V} = \mathbf{C}^{-1}. \tag{7.26}$$

Definition 7.9 — Model-view matrix. The **model view matrix** is the model matrix relative to the camera:

$$\text{ModelView} = \mathbf{VM} = \mathbf{C}^{-1}\mathbf{M}. \tag{7.27}$$

Summary 7.6 In OpenGL, we store vertex postions in a geometry spreadsheet as arrays of numbers (either in $\mathbf{p}_{3D}$ or $\mathbf{p}_{4D}$). We do not modify these numbers even when the object has been transformed. When we transform the object, we modify its model matrix.

The model matrices are managed by a **matrix stack** that we will talk about in a later chapter.

A vertex shader will have uniform variables that allow us to send the model matrix and the view matrix (or just the model-view matrix), so that the vertex positions can be transformed into camera's coordinate frame in the vertex shader.

7.4.1 "Look-at" camera/view matrix

Definition 7.10 A camera's coordinate system $\begin{bmatrix} C\vec{\mathbf{e}} & C\underline{o} \end{bmatrix}$ is defined so that

- $C\vec{\mathbf{e}}$ is orthonormal.
- $C\underline{o}$ is at the center of the camera.
- $C\vec{e}_x$ is pointing to the right of the camera.

- $\vec{Ce}_y$ is pointing towards the top of the camera.
- $\vec{Ce}_z = \vec{Ce}_x \times \vec{Ce}_y$ is pointing to the back of the camera.

How do we set up these coordinate system intuitively? A common technique is to describe the camera in terms of

- The target the camera looks at;
- The position of the camera (also known as the eye) ($C\underline{o}$);
- The up-vector ($\vec{Ce}_y$).

These parameters are more intuitive if we want to maneuver the camera.

Task 7.1 Let $\begin{bmatrix}\vec{\mathbf{e}} & \underline{o}\end{bmatrix}$ be a world coordinate system (with orthonormal $\vec{\mathbf{e}}$). Let

- $\vec{\mathbf{e}}\mathbf{i} + \underline{o}$, $\mathbf{i} \in \mathbb{R}^3$, be the desired eye position;
- $\vec{\mathbf{e}}\mathbf{c} + \underline{o}$, $\mathbf{c} \in \mathbb{R}^3$, be the position of the desired target camera looks at;
- $\vec{\mathbf{e}}\mathbf{u}$, $\mathbf{u} \in \mathbb{R}^3$, be the desired up vector;

find the view matrix $\mathbf{V} \in \mathbb{R}^{4\times4}$, which is the inverse of camera's model matrix $\mathbf{C}$.

That is, build the helper function

$$\mathbf{V} = \mathrm{LookAt}(\mathbf{i}, \mathbf{c}, \mathbf{u}). \tag{7.28}$$

Note that the up vector might not be a valid up vector: the up vector $\vec{Ce}_y = \vec{\mathbf{e}}\mathbf{u}$ must be perpendicular to the axis $\vec{Ce}_z$, and $\vec{Ce}_z$ has already been determined by the displacement between the target and the eye:

$$\vec{Ce}_z = \vec{\mathbf{e}}\mathbf{z} = \vec{\mathbf{e}}\frac{\mathbf{i}-\mathbf{c}}{|\mathbf{i}-\mathbf{c}|}, \quad \text{or simply } \mathbf{z} = \frac{\mathbf{i}-\mathbf{c}}{|\mathbf{i}-\mathbf{c}|}. \tag{7.29}$$

So, we first update $\mathbf{u}$ so that it is absolutely orthogonal to $\mathbf{z}$:

$$\mathbf{u}^{\text{orthogonalized}} = \mathbf{u} - (\mathbf{z}\cdot\mathbf{u})\mathbf{z} \tag{7.30}$$

$$\mathbf{u}^{\text{updated}} = \frac{\mathbf{u}^{\text{orthogonalized}}}{|\mathbf{u}^{\text{orthogonalized}}|}. \tag{7.31}$$

In an implementation of the LookAt function, it is good to have this updating step for the up vector $\mathbf{u}$. We will drop the superscript of $\mathbf{u}^{\text{updated}}$, assuming that $\mathbf{u}$ has already been updated.

Now, we have established $\vec{Ce}_z = \vec{\mathbf{e}}\mathbf{z}$, $\vec{Ce}_y = \vec{\mathbf{e}}\mathbf{u}$, and $C\underline{o} = \vec{\mathbf{e}}\mathbf{i} + \underline{o}$. The only remaining coordinate axis is:

$$\vec{Ce}_x = \vec{Ce}_y \times \vec{Ce}_z = \vec{\mathbf{e}}\,(\mathbf{u}\times\mathbf{z}), \quad \text{or simply } \mathbf{x} = \mathbf{u}\times\mathbf{z} \text{ for } \vec{Ce}_x = \vec{\mathbf{e}}\mathbf{x}. \tag{7.32}$$

Then, the camera matrix $\mathbf{C}$, which satisfies

$$\begin{bmatrix}\vec{\mathbf{e}}\mathbf{x} & \vec{\mathbf{e}}\mathbf{u} & \vec{\mathbf{e}}\mathbf{z} & (\vec{\mathbf{e}}\mathbf{i}+\underline{o})\end{bmatrix} = \begin{bmatrix}\vec{Ce}_x & \vec{Ce}_y & \vec{Ce}_z & C\underline{o}\end{bmatrix} = \begin{bmatrix}\vec{\mathbf{e}} & \underline{o}\end{bmatrix}\mathbf{C}, \tag{7.33}$$

is

$$\mathbf{C} = \begin{bmatrix} | & | & | & | \\ \mathbf{x} & \mathbf{u} & \mathbf{z} & \mathbf{i} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7.34}$$

Now, $\mathbf{V} = \mathbf{C}^{-1}$. Note that the 3-by-3 block $\mathbf{C}_{3\times3} = \begin{bmatrix} | & | & | \\ \mathbf{x} & \mathbf{u} & \mathbf{z} \\ | & | & | \end{bmatrix}$ is a *rotation matrix*. In particular,

$$\mathbf{C}_{3\times3}^{-1} = \mathbf{C}_{3\times3}^{\mathsf{T}} = \begin{bmatrix} — & \mathbf{x}^{\mathsf{T}} & — \\ — & \mathbf{u}^{\mathsf{T}} & — \\ — & \mathbf{z}^{\mathsf{T}} & — \end{bmatrix}. \tag{7.35}$$

Then one can check that

$$\mathbf{V} = \mathbf{C}^{-1} = \begin{bmatrix} \mathbf{C}_{3\times3}^{-1} & -\mathbf{C}_{3\times3}^{-1}\mathbf{i} \\ \mathbf{0} & 1 \end{bmatrix}. \tag{7.36}$$

7.5 Transformation of normal vectors

The vertex normal is often one of the vertex attributes that we read in the vertex shader. Suppose our model-view matrix is $\mathbf{M}$, how should we transform the normal vector $\mathbf{n}$ to the correct array of numbers that make sense in the camera's coordinate system? If

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix} \tag{7.37}$$

and $\mathbf{n} \in \mathbb{R}^3$, should we take the product $\mathbf{An}$? The answer would be yes if $\mathbf{n}$ is a *displacement* type of vector. However, normal vectors transform differently.

We want to transform $\mathbf{n}$ so that the transformed $\mathbf{n}'$ is a normal vector to the geometry transformed by $\mathbf{M}$. That is, suppose $\mathbf{u}$ is an arbitrary tangent vector to the surface (*i.e.*$\mathbf{u}^{\mathsf{T}}\mathbf{n} = 0$). Then we want $(\mathbf{Au})^{\mathsf{T}}\mathbf{n}' = 0$. This condition says that

$$\mathbf{u}^{\mathsf{T}}\mathbf{A}^{\mathsf{T}}\mathbf{n}' = 0 \quad \text{for all } \mathbf{u} \text{ so that } \mathbf{u}^{\mathsf{T}}\mathbf{n} = 0, \tag{7.38}$$

implying that $\mathbf{A}^{\mathsf{T}}\mathbf{n}'$ must be a scalar multiple of $\mathbf{n}$:

$$\mathbf{A}^{\mathsf{T}}\mathbf{n}' = \alpha\mathbf{n} \tag{7.39}$$

for some scalar α. Therefore,

$$\mathbf{n}' = \alpha\mathbf{A}^{-\mathsf{T}}\mathbf{n} \tag{7.40}$$

The scalar α does not matter. We can just transform the normal by $\mathbf{A}^{-\mathsf{T}}\mathbf{n}$ and then followed by a normalization (as we usually want the normals to be unit vectors).

Summary 7.7 In a vertex shader, suppose we have read a vertex position $\mathbf{p} \in \mathbb{R}^4$, displacement $\mathbf{u} \in \mathbb{R}^3$, and normal vector $\mathbf{n}$. Suppose the model-view matrix is given by $\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0} & 1 \end{bmatrix}$. Then we transform

$$\mathbf{p} \mapsto \mathbf{Mp}, \quad \mathbf{u} \mapsto \mathbf{Au}, \quad \mathbf{n} \mapsto \mathbf{A}^{-\mathsf{T}}\mathbf{n}. \tag{7.41}$$

Then we normalize the normal $\mathbf{n} \mapsto \mathbf{n}/|\mathbf{n}|$.

7.6 Exercise: a Camera class for a model viewer

Let us implement Section 7.4.1 into a class called `Camera`. A camera is an object that has the information to determine the view matrix (the inverse of the model matrix relative to the world). They are the parameters for the "Look-At" function.

7.6.1 Camera class

An sample header file for the camera class is given by:

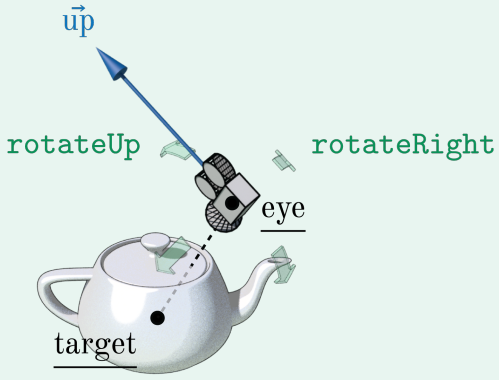
Code 7.1 `Camera.h`

```
/*************************************************
Camera is a class for a camera object.
*************************************************/
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>

#ifndef __CAMERA_H__
#define __CAMERA_H__

class Camera {
public:
    glm::vec3 eye;// position of the eye
    glm::vec3 target;  // look at target
    glm::vec3 up;      // up vector
    // Ohter parameters that we will cover in the next section
    //float fovy;  // field of view in degrees
    //float aspect; // aspect ratio
    //float near; // near clipping distance
    //float far; // far clipping distance

    // default values for reset
    glm::vec3 eye_default = glm::vec3(5.0f, 0.0f, 0.0f);
    glm::vec3 target_default = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec3 up_default = glm::vec3(0.0f, 1.0f, 0.0f);
    // Ohter parameters that we will cover in the next section
    //float fovy_default = 30.0f;
    //float aspect_default = 4.0f/3.0f;
    //float near_default = 0.01f;
    //float far_default = 100.0f;

    glm::mat4 view = glm::mat4(1.0f);    // view matrix
    glm::mat4 proj = glm::mat4(1.0f);    // projection matrix

    void rotateRight(const float degrees);
    void rotateUp(const float degrees);
    void computeMatrices(void);
    void reset(void);
};

#endif
```

Note that there are some other parameters involving the perspective projection, which is what we will learn about in the next chapter.

Exercise 7.1 Implement

- `Camera::rotateRight(const float degrees)`: Update the variables `eye`, `target` and `up`.
- `Camera::rotateUp(const float degrees)`: Update the variables `eye`, `target` and `up`.
- `Camera::computeMatrices(void)`: Update the matrix `view` so that it is the view matrix of the current configuration.



Both the `rotateRight` and `rotateUp` will not modify the `target`. The function `rotateRight` rotates the camera position `eye` about an axis through the `target` parallel to the normalized `up` vector. The function `rotateUp` rotates the camera position `eye` and the `up` vector about an axis through the `target`; this axis is orthogonal to both the normalized `up` vector and the displacement between the `eye` and the `target`. How much these rotation functions will rotate are given by the angle (in degrees) of the input argument.

Code 7.2 `Camera.cpp`

```
#include "Camera.h"
#include <math.h>

glm::mat3 rotation(const float degrees,const glm::vec3 axis){
    // from the previous section
}

void Camera::rotateRight(const float degrees){
    // TODO
}
void Camera::rotateUp(const float degrees){
    // TODO
}

void Camera::computeMatrices(){
```

```
    // Note that glm matrix column majored.
    // That is, A[i] is the ith column of A,
    // and A_{ij} in math notation is A[j][i] in glm notation.

    // VIEW MATRIX
      // TODO

    // PROJECTION MATRIX
      // TODO next time
}

void Camera::reset(){
    eye = eye_default;// position of the eye
    target = target_default;  // look at target
    up = up_default;      // up vector
    fovy = fovy_default;  // field of view in degrees
    aspect = aspect_default; // aspect ratio
    near = near_default; // near clipping distance
    far = far_default; // far clipping distance
}
```

We can integrate this camera into a "model viewer" application.

7.6.2 Model viewer (main.cpp)

We can create a simple program that places a 3D geometry (*e.g.* a cube) and have the camera looking at it.

Code 7.3 `main.cpp` for the ModelViewer application.

```
#include <stdlib.h>
#include <iostream>

#ifdef __APPLE__
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
#else
#include <GL/glew.h>
#include <GL/glut.h>
#endif
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>

#include "Shader.h"
#include "Cube.h"
#include "Camera.h"


static const int width = 800;
static const int height = 600;
static const char* title = "Model viewer";
static const glm::vec4 background(0.1f, 0.2f, 0.3f, 1.0f); // bg color
static Cube cube;     // this is our geometric object to view
static Camera camera; // the camera object
```

```
25
26  // A simple shader that has uniforms of the modelview matrix and a projection
        matrix (we will learn about the projection matrix in the next section).
27  struct NormalShader : Shader {
28
29      // modelview and projection
30      glm::mat4 modelview = glm::mat4(1.0f); GLuint modelview_loc;
31      glm::mat4 projection = glm::mat4(1.0f); GLuint projection_loc;
32
33      void initUniforms(){
34          modelview_loc = glGetUniformLocation( program, "modelview" );
35          projection_loc = glGetUniformLocation( program, "projection" );
36      }
37      void setUniforms(){
38          glUniformMatrix4fv(modelview_loc, 1, GL_FALSE, &modelview[0][0]);
39          glUniformMatrix4fv(projection_loc, 1, GL_FALSE, &projection[0][0]);
40      }
41  };
42  static NormalShader shader;
43  static bool enable_perspective = false;
44  // A simple projection matrix when we do not have a projection matrix.
45  static glm::mat4 proj_default = glm::mat4(0.75f,0.0f,0.0f,0.0f,
46                                            0.0f,1.0f,0.0f,0.0f,
47                                            0.0f,0.0f,-0.1f,0.0f,
48                                            0.0f,0.0f,0.0f,1.0f);
49
50  void printHelp(){
51      std::cout << R"(
52      Available commands:
53        press 'h' to print this message again.
54        press Esc to quit.
55        press the arrow keys to rotate camera.
56        press 'r' to reset camera.
57        press 'p' to toggle orthographic/perspective.
58  )";
59      std::cout<< "perspective: " << (enable_perspective?"on":"off") << std::endl
        ;
60  }
61
62  void initialize(void){
63      printHelp();
64      glClearColor(background[0], background[1], background[2], background[3]);
        // background color
65      glViewport(0,0,width,height);
66
67      // Initialize geometries
68      cube.init(); // this will set up the vertex array object for the cube
69
70      // Initialize camera (set default values)
71      camera.eye_default = glm::vec3(0.0f, 0.2f, 5.0f);
72      camera.target_default = glm::vec3(0.0f, 0.2f, 0.0f);
73      camera.up_default = glm::vec3(0.0f, 1.0f, 0.0f);
74      //camera.fovy_default = 30.0f;
75      //camera.aspect_default = float(width) / float(height);
76      //camera.near_default = 0.01f;
```

```
77      //camera.far_default = 100.0f;
78      camera.reset();
79
80      // Initialize shader
81      shader.read_source( "shaders/projective.vert", "shaders/normal.frag");
82      shader.compile();
83      glUseProgram(shader.program);
84      shader.initUniforms();
85      shader.setUniforms();
86
87      // Enable depth test
88      glEnable(GL_DEPTH_TEST);
89  }
90
91  void display(void){
92      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
93
94      // Pre-draw sequence
95      camera.computeMatrices();
96      shader.modelview = camera.view;
97      shader.projection = enable_perspective? camera.proj : proj_default;
98      shader.setUniforms();
99      // BEGIN draw
100     cube.draw();
101     // END draw
102
103     glutSwapBuffers();
104     glFlush();
105
106 }
107
108 void keyboard(unsigned char key, int x, int y){
109     switch(key){
110         case 27: // Escape to quit
111             exit(0);
112             break;
113         case 'h': // print help
114             printHelp();
115             break;
116         case 'o': // save screenshot
117             saveScreenShot();
118             break;
119         case 'r': // reset camera
120             camera.reset();
121             glutPostRedisplay();
122             break;
123         case 'p': // toggle perspective view
124             enable_perspective = !enable_perspective;
125             std::cout<< "perspective: " << (enable_perspective?"on":"off") <<
    std::endl;
126             glutPostRedisplay();
127             break;
128         default:
129             glutPostRedisplay();
130             break;
```

```
131     }
132 }
133 void specialKey(int key, int x, int y){
134     switch (key) {
135     // keyboard direction and camera movements are opposite to each other for
        the "reversed scrolling" effect.
136         case GLUT_KEY_UP: // up
137             camera.rotateUp(-10.0f);
138             glutPostRedisplay();
139             break;
140         case GLUT_KEY_DOWN: // down
141             camera.rotateUp(10.0f);
142             glutPostRedisplay();
143             break;
144         case GLUT_KEY_RIGHT: // right
145             camera.rotateRight(-10.0f);
146             glutPostRedisplay();
147             break;
148         case GLUT_KEY_LEFT: // left
149             camera.rotateRight(10.0f);
150             glutPostRedisplay();
151             break;
152     }
153 }
154
155 int main(int argc, char** argv)
156 {
157     // BEGIN CREATE WINDOW
158     glutInit(&argc, argv);
159
160 #ifdef __APPLE__
161     glutInitDisplayMode( GLUT_3_2_CORE_PROFILE | GLUT_DOUBLE | GLUT_RGBA |
        GLUT_DEPTH);
162 #else
163     glutInitContextVersion(3,1);
164     glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH );
165 #endif
166     glutInitWindowSize(width, height);
167     glutCreateWindow(title);
168 #ifndef __APPLE__
169     glewExperimental = GL_TRUE;
170     GLenum err = glewInit() ;
171     if (GLEW_OK != err) {
172         std::cerr << "Error: " << glewGetErrorString(err) << std::endl;
173     }
174 #endif
175     std::cout << "OpenGL Version: " << glGetString(GL_VERSION) << std::endl;
176     // END CREATE WINDOW
177
178     initialize();
179     glutDisplayFunc(display);
180     glutKeyboardFunc(keyboard);
181     glutSpecialFunc(specialKey);
182
183     glutMainLoop();
```

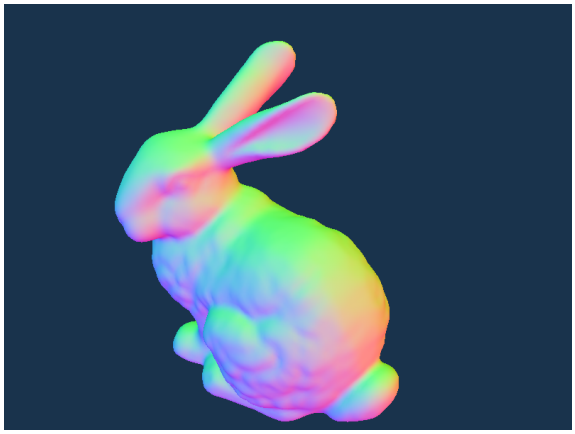


Figure 7.2 The color as a function of the normal vector based on formula (7.42).

```
184         return 0;
185 }
```

7.6.3 A simple normal shader

We need to find a way to color the 3D geometry so that it looks 3D. This is typically done by lighting. Since we have not talked about lighting, we will use a simple "normal shader." That is, we just assign the color as a function of the normal vector.

A normal vector $\mathbf{n}$ lie on the unit sphere centered at the origin $\mathbb{S}^2 = \{(x, y, z) \,|\, x^2 + y^2 + z^2 = 1\}$. The color space is a 3D vector (RGB) in the $[0, 1]^3$ box. The standard normal shader simply shifts and scales this unit sphere so that it fits into the box. See Figure 7.2.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \frac{1}{2} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} + \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}. \tag{7.42}$$

The following are the vertex shader and the fragment shader.

Code 7.4 `projective.vert`

```
1  # version 330 core
2
3  layout (location = 0) in vec3 vertex_position;
4  layout (location = 1) in vec3 vertex_normal;
5
6  uniform mat4 modelview;
7  uniform mat4 projection;
8
9  out vec4 position;
10 out vec3 normal;
11
12 void main(){
13     gl_Position = projection * modelview * vec4( vertex_position, 1.0f );
14     // forward the raw position and normal in the model coord to frag shader
15     position = vec4(vertex_position, 1.0f );
16     normal = vertex_normal;
```

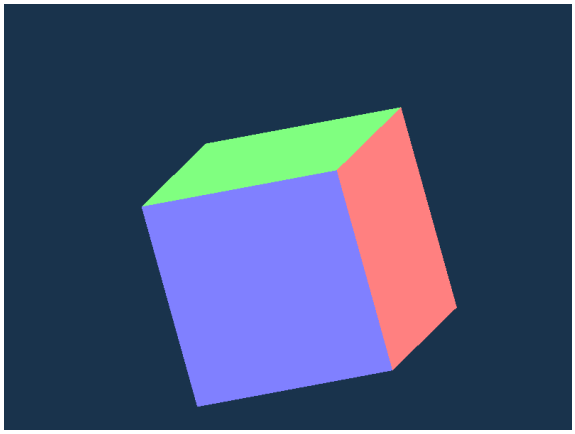


Figure 7.3 The result of ModelViewer. Note that the camera target is set slightly off from the cube center (world coordinate origin). This setup allows us to test if there is no bug in preserving the camera target during the camera rotation.

```
}
```

Code 7.5 `normal.frag`

```
#version 330 core

in vec4 position;
in vec3 normal;

uniform mat4 modelview;

// Output the frag color
out vec4 fragColor;


void main (void){
    vec3 N = normalize(normal);
    fragColor = vec4(0.5f*N + 0.5f , 1.0f);
}
```

7.6.4 Geometry class

The only remaining part of the model viewer is the cube geometry.

Let us update our geometry class a bit so that it has virtual member functions. The "`virtual`" keyword is important when a scene contains many different derived classes (cube, sphere, *etc.*) of this base Geometry class. In that scenario, we want to maintain a container (array, vector, map, *etc.*) of pointers that refer to these various geometry subclass objects. Naturally, the container would be of the type `std::vector< Geometry* >`, containing the pointers to the base class, so that it works as a container for all derived classes. However, the pointers will degenerate into pointers pointing to the base class, rather than the derived class. If we call a function from one of these geometries, we want to make sure that the compiler actually calls the function that is implemented (overridden) in the derived class, rather than the

default implementation by the base class. This is achieved by adding the "`virtual`" keyword in front of those function. A function is `virtual` if calling the derived class implementation is of higher priority.

We also *do not* make the functions *purely virtual* (with the "equals zero" in the declaration `virtual T memberFunction(...)=0;`) because we don't want to require all derived classes to implement every single function.

Code 7.6 `Geometry.h`

```
#include <vector>

#ifndef __GEOMETRY_H__
#define __GEOMETRY_H__

class Geometry {
public:
    GLenum mode = GL_TRIANGLES; // the cookboook for glDrawElements
    int count; // number of elements to draw
    GLenum type = GL_UNSIGNED_INT; // type of the index array
    GLuint vao; // vertex array object a.k.a. geometry spreadsheet
    std::vector<GLuint> buffers; // data storage

    // methods to call during initialization (implemented in each derived class
    )
    virtual void init(){};
    virtual void init(const char* s){}; // an option for reading files from a
    given filename s.

    // the draw command to call during display.  It's a fixed function so there
     is not a "virtual" keyword.
    void draw(void){
        glBindVertexArray(vao);
        glDrawElements(mode,count,type,0);
    }
};

#endif
```

7.6.5 Cube: a derived class from the Geometry class

We will configure the vertex attributes to consist of positions and normals.

We just implement one standard cube that is centered at the origin with side lengths equal one. If the users want a cube (or even a rectangular box) with a different side lengths and position, they handle it by transformations down the pipeline.

Code 7.7 `Cube.h`

```
/*************************************************
Cube is subclass class of Geometry
that represents a 3D cube.
*************************************************/
#include "Geometry.h"
```

```
6  #ifndef __CUBE_H__
7  #define __CUBE_H__
8
9  class Cube : public Geometry {
10 public:
11   void init(void){
12     // vertex positions
13     const GLfloat positions[24][3] ={
14         // Front face
15         { -0.5f, -0.5f, 0.5f },{ -0.5f, 0.5f, 0.5f },{ 0.5f, 0.5f, 0.5f },{ 0.5
       f, -0.5f, 0.5f },
16         // Back face
17         { -0.5f, -0.5f, -0.5f },{ -0.5f, 0.5f, -0.5f },{ 0.5f, 0.5f, -0.5f },{
       0.5f, -0.5f, -0.5f },
18         // Left face
19         { -0.5f, -0.5f, 0.5f },{ -0.5f, 0.5f, 0.5f },{ -0.5f, 0.5f, -0.5f },{
       -0.5f, -0.5f, -0.5f },
20         // Right face
21         { 0.5f, -0.5f, 0.5f },{ 0.5f, 0.5f, 0.5f },{ 0.5f, 0.5f, -0.5f },{ 0.5f,
        -0.5f, -0.5f },
22         // Top face
23         { 0.5f, 0.5f, 0.5f },{ -0.5f, 0.5f, 0.5f },{ -0.5f, 0.5f, -0.5f },{ 0.5
       f, 0.5f, -0.5f },
24         // Bottom face
25         { 0.5f, -0.5f, 0.5f },{ -0.5f, -0.5f, 0.5f },{ -0.5f, -0.5f, -0.5f },{
       0.5f, -0.5f, -0.5f }
26     };
27     // vertex normals
28     const GLfloat normals[24][3] = {
29         // Front face
30         { 0.0f, 0.0f, 1.0f },{ 0.0f, 0.0f, 1.0f },{ 0.0f, 0.0f, 1.0f },{ 0.0f,
       0.0f, 1.0f },
31         // Back face
32         { 0.0f, 0.0f, -1.0f },{ 0.0f, 0.0f, -1.0f },{ 0.0f, 0.0f, -1.0f },{ 0.0
       f, 0.0f, -1.0f },
33         // Left face
34         { -1.0f, 0.0f, 0.0f },{ -1.0f, 0.0f, 0.0f },{ -1.0f, 0.0f, 0.0f },{
       -1.0f, 0.0f, 0.0f },
35         // Right face
36         { 1.0f, 0.0f, 0.0f },{ 1.0f, 0.0f, 0.0f },{ 1.0f, 0.0f, 0.0f },{ 1.0f,
       0.0f, 0.0f },
37         // Top face
38         { 0.0f, 1.0f, 0.0f },{ 0.0f, 1.0f, 0.0f },{ 0.0f, 1.0f, 0.0f },{ 0.0f,
       1.0f, 0.0f },
39         // Bottom face
40         { 0.0f, -1.0f, 0.0f },{ 0.0f, -1.0f, 0.0f },{ 0.0f, -1.0f, 0.0f },{ 0.0
       f, -1.0f, 0.0f }
41     };
42     // Cube indices
43     const GLuint indices[36] = {
44         0, 1, 2, 0, 2, 3, // Front face
45         4, 5, 6, 4, 6, 7, // Back face
46         8, 9, 10, 8, 10, 11, // Left face
47         12, 13, 14, 12, 14, 15, // Right face
48         16, 17, 18, 16, 18, 19, // Top face
```

```
49          20, 21, 22, 20, 22, 23 // Bottom face
50      };
51      glGenVertexArrays(1, &vao );
52      buffers.resize(3); // recall that buffers is std::vector<GLuint>
53      glGenBuffers(3, buffers.data());
54      glBindVertexArray(vao);
55
56      // 0th attribute: position
57      glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
58      glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW)
        ;
59      glEnableVertexAttribArray(0);
60      glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,(void*)0);
61
62      // 1st attribute: normal
63      glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
64      glBufferData(GL_ARRAY_BUFFER, sizeof(normals), normals, GL_STATIC_DRAW);
65      glEnableVertexAttribArray(1);
66      glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,0,(void*)0);
67
68      // indices
69      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
70      glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
        GL_STATIC_DRAW);
71
72      count = sizeof(indices)/sizeof(indices[0]);
73      glBindVertexArray(0);
74    }
75
76
77  };
78
79  #endif
```

8. Projective Geometry

As the final step in the vertex shader, we transform the objects to a configuration, so that the rasterizer can give us a desired 2D image. What is this configuration that we want? The question is asked by the Renaissant artists in the 15th century: *What is a proper 2D representation of a 3D object?* A precise approach to the question was developed by the artists, called the technique of *perspective drawing*. The mathematics behind the artistic technique is **projective geometry**, which lies at the very foundation to every perspective view in 3D computer graphics.

8.1 Projections in graphics

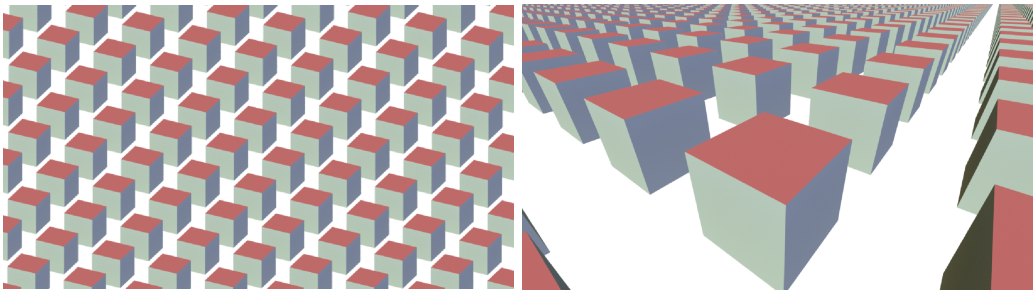
In Section 7.4, we learned about how to represent any position written in any coordinate frame as an array of numbers in the camera's coordinate system. This is achieved by applying an affine *model-view* matrix to the position from model's coordinate system.

In this chapter, we study the final transformation that transforms objects from the **camera coordinate system** to the **normalized device coordinate** (*cf.* Definition 3.3).

Definition 8.1 The normalized device coordinate (NDC) has a **viewing box**, also called a **clipping box**, given by $[-1,1]\times[-1,1]\times[-1,1]\subset\mathbb{R}^3$. Note that the coordinate axes in the normalized device coordinate is such that

- $\vec{e}_x^{\text{NDC}}$ points to the right, representing the right direction of the image;
- $\vec{e}_y^{\text{NDC}}$ points up, representing the up direction of the image;
- $\vec{e}_z^{\text{NDC}}$ points *forward*, *i.e.* $\vec{e}_z=-\vec{e}_x\times\vec{e}_y$, so that it represents the *depth*. (It is a left-handed basis.)

By reading the x,y coordinates of a position, we have its location in the image. By comparing the z value, we have the correct occlusion relation among fragments.



(a) Orthographic (affine) projection. (b) Perspective projection.

Figure 8.1 Orthographic and perspective projections.

Definition 8.2 The transformation from the camera coordinate system to the normalized device coordinate (NDC) is called a **projection**. Although it is still mapping a 3D space to a 3D space (as opposed to mapping a 3D space down to a 2D space as what the word "projection" usually refers to), we call it a projection and understand the final z coordinate as just an additional depth information.

A simple transformation can simply be shrinking and translating (affine transformation) the objects from the camera frame so that they fit in the clipping box in the NDC. This projection is called an **orthographic projection**, also called an **affine projection**, **isometric view**, or an **oblique projection**.

In most cases, however, we want a transformation that is beyond affine transformations. We want to be able to get a **perspective projection**. See Figure 8.1.

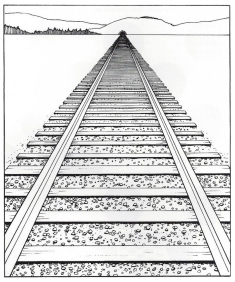
8.1.1 Orthographic projection

An orthographic projection is an affine transformation.

- As any affine transformation, it preserves the notion of parallelism. If there are parallel lines in the original space, then the lines still appear parallel in the image.
- The scaling of objects is uniform. Further object does not look smaller.
- The image lacks a sense of depth.
- Orthographic views show up as the telescopic views from a vantage point far away.

8.1.2 Perspective projection

We will focus on perspective transformation in this chapter. They are non-affine, but it does preserve collinearity.



- If three points are collinear in the original space, then they are still collinear in the image.
- Parallelism is generally not preserved. If there are two parallel lines in the original space, then it is possible that the two lines intersect in the image. If the two parallel lines are parallel to the ground, then they meet at the so-called **vanishing point** on the **horizon** of the ground in a perspective image.

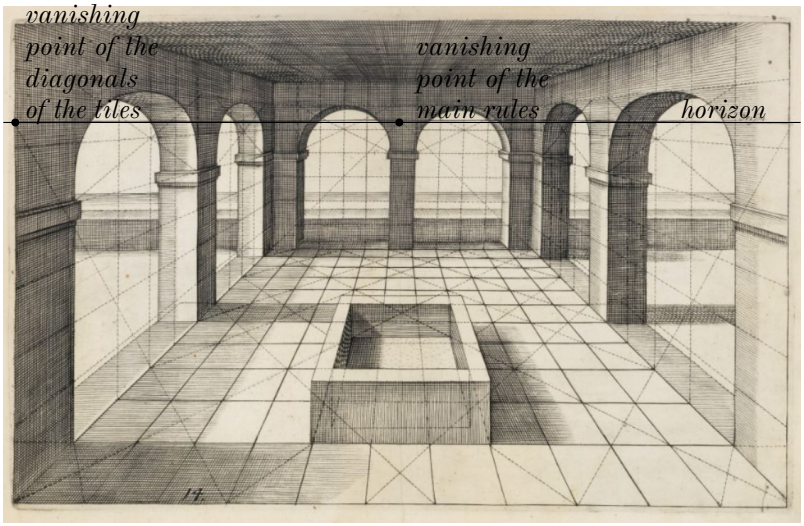


Figure 8.2 Linear perspective drawing of a square tiling over the floor.

- The scaling of objects is non-uniform. Further objects look smaller, which is a phenomenon called **foreshortening**.
- Perspective views provide a sense of depth. They give the image viewer a more immersive experience.

8.2 From Renaissant arts to projective geometry

8.2.1 Perspective drawing

Filippo Brunelleschi (1337–1446) introduced the revolutionary technique of **linear perspective** in 1415, which is the perspectivity we talk about here. The principle of linear perspectivity is that *further objects appear smaller in the image, while straight lines stay straight*. Soon after Brunelleschi, many artists in Florence, including Leon Battista Alberti and Leonardo da Vinci, adopted and further developed the method of linear perspective.

The drawing technique features **vanishing points**, which represent the intersections of sets of parallel lines in the original 3D space. A line that joins a point A to a vanishing point would depict a line that passes through A and is parallel to an axis or a direction represented by the vanishing point. A good exercise is to use this principle repeatedly to construct a square tiling over the floor like Figure 8.2. In particular, with the aid of the vanishing point of the diagonal lines of the tiles, we can construct the correct perspective distortion of equally spaced marks along a rule.

8.2.2 Elements at infinity

Mathematician Girard Desargues (1591–1661), usually regarded as the founder of projective geometry, realized that the linear perspective art is governed by a geometric theory about lines. In this new theory, parallel lines have an intersection point, called points at infinity. This intersection does appear in an image after a perspective transformation, and thus we must embrace these points at infinity in the original space, otherwise perspective transformations would create or annihilate

geometric elements rather than only permuting elements. The spaces including additional elements at infinity are called **projective spaces**.

Definition 8.3 The **real projective line** $\mathbb{RP}^1$ is the real line $\mathbb{R}$ with one point called infinity

$$\mathbb{RP}^1 = \mathbb{R} \cup \{\infty\}. \tag{8.1}$$

The **real projective plane** $\mathbb{RP}^2$ is the plane $\mathbb{R}^2$ with an additional projective line called the line at infinity (horizon)

$$\mathbb{RP}^2 = \underbrace{\mathbb{R}^2}_{\text{finite points}} \cup \underbrace{\mathbb{RP}^1}_{\text{line at infinity}} . \tag{8.2}$$

Similarly, the **real projective space** $\mathbb{RP}^3$ is the space $\mathbb{R}^3$ with an additional projective plane called the plane at infinity (sky)

$$\mathbb{RP}^3 = \underbrace{\mathbb{R}^3}_{\text{finite points}} \cup \underbrace{\mathbb{RP}^1}_{\text{plane at infinity}} . \tag{8.3}$$

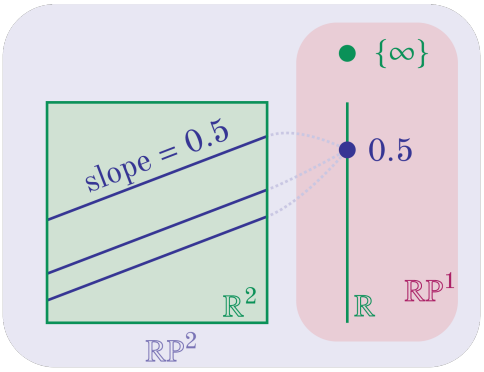


Figure 8.3 Desargues' projective plane is an extension of the standard $\mathbb{R}^2$ plane with a line at infinity, so that even parallel lines have an intersection.

For instance, in the projective plane, any two distinct lines have one and only one intersection. If these two lines are not parallel, then they have an intersection at a finite point in $\mathbb{R}^2$. If the two lines are parallel, then they have an intersection marked on $\mathbb{RP}^1$. Each parallel family of lines have the same slope. So each element in $\mathbb{RP}^1 = \mathbb{R} \cup \{\infty\}$ is marking the slope. The slope of ∞ corresponds to the vertical slope. (Figure 8.3.)

However, Desargues' theory with infinities was not fully understood by the people at his time.

8.2.3 Taylor's Principles of Linear Perspective

Brook Taylor gave a more understandable form of the principles of linear perspective in 1715. Taylor spelled out the perspective painting techniques mathematically. It is

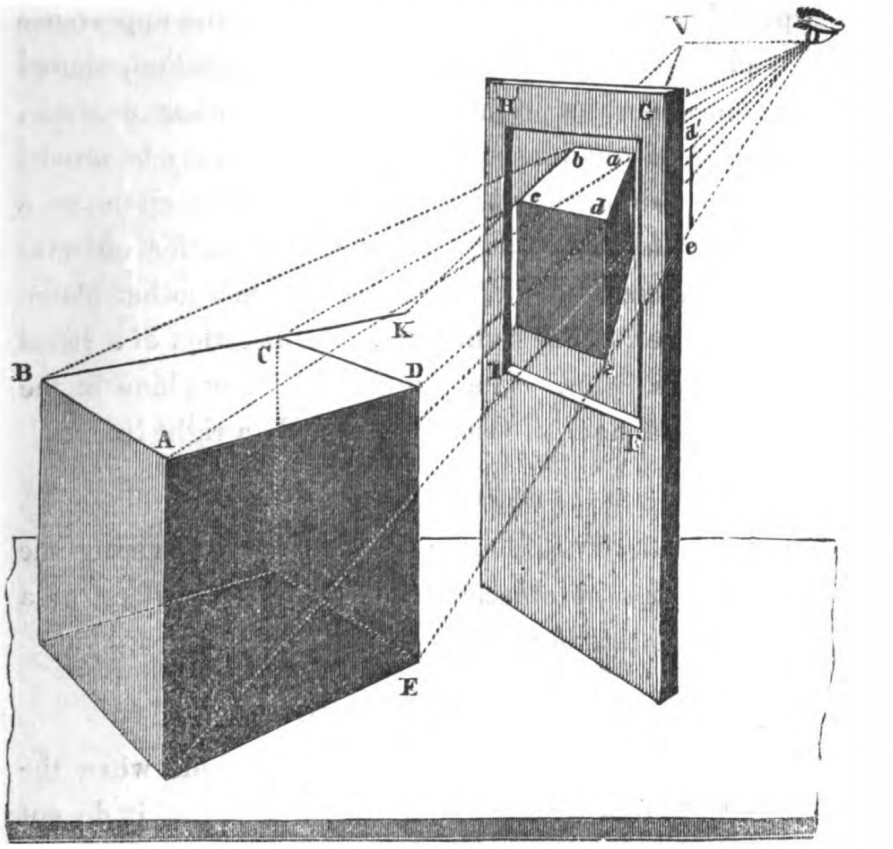


Figure 8.4 Taylor's *Principles of Linear Perspective* (1715).

the straight light rays passing through one point, called the eye, that are the image which we see. We can take a plane (canvas) to intercept the lines to form the image. "*The light ought to come from the Picture to the spectator's Eye in the very same manner as it would from the objects themselves.*" The artistic/Desarguesian notions of horizon, vanishing points, infinity, *etc.*, happen in this canvas. This canvas would be Desargues' projective plane. The geometric theories in the projective planes (2D) can be understood as a shadow casted from the higher-dimensional (3D) but standard geometry.

8.2.4 Homogeneous coordinates

The projective geometry of Desargues came to its full glory as August Möbius introduced the **homogeneous coordinates** in 1827. The idea is similar to Taylor's picture. *The projective geometry is merely the linear algebra in the vector space one dimension higher (with the eye being the origin).*

To describe lines passing through the origin, we use the notion of *proportion*.

Definition 8.4 Two arrays of numbers (vectors) $\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \neq \mathbf{0}$ and $\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \neq \mathbf{0}$ are said to

have the same proportion if one is the scale of the other one. That is, there exists $\lambda \in \mathbb{R}$, $\lambda \neq 0$, so that

$$\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \lambda x_1 \\ \vdots \\ \lambda x_n \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{8.4}$$

In this case, we denote

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \sim \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{8.5}$$

If we collect all vectors of the same proportion, then they form a straight line passing through the origin. If we think of the eye as the origin, each light ray through the eye is a proportion.

Definition 8.5 The projective line $\mathbb{RP}^1$ is the set of proportions of $\mathbb{R}^2$ vectors. The projective plane $\mathbb{RP}^2$ (such as the canvas in Figure 8.4) is the set of proportions of $\mathbb{R}^3$ vectors (the eye is the origin of the $\mathbb{R}^3$ space). Similarly, the projective space $\mathbb{RP}^3$ is the set of proportions of $\mathbb{R}^4$ vectors.

One can see that Definition 8.5 is consistent with Definition 8.3 with the following process. Let us take the projective plane $\mathbb{RP}^2$ as an example.

- **Homogenization**: For each point $\mathbf{p}_{2D} = \begin{bmatrix} p_x \\ p_y \end{bmatrix} \in \mathbb{R}^2$, we construct

$$\mathbf{p}_{3D} = \begin{bmatrix} \mathbf{p}_{2D} \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \tag{8.6}$$

  and call it the homogeneous coordinate for $\mathbf{p}_{2D}$.

- **Dehomogenization**: Given any nonzero vector $\mathbf{p}_{3D} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \in \mathbb{R}^3$, we rescale the vector (which still represents the same proportion) so that the last component is 1,

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \sim \begin{bmatrix} p_x/p_z \\ p_y/p_z \\ 1 \end{bmatrix}, \tag{8.7}$$

  and then we can read off the 2D component:

$$\mathbf{p}_{2D} = \begin{bmatrix} p_x/p_z \\ p_y/p_z \end{bmatrix}. \tag{8.8}$$

- **Infinity elements**: If the 3D vector $\mathbf{p}_{3D} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \in \mathbb{R}^3$ has the last component $p_z = 0$, then we cannot dehomogenize the vector due to division by zero. In

that case, we just take the element as is $\begin{bmatrix} p_x \\ p_y \\ 0 \end{bmatrix}$ (which is an element in $\mathbb{RP}^1$ since it is a proportion of $\mathbb{R}^2$ vector) and say it is a point on the line at infinity.

Note 8.1 *In Chapter 7, we have been representing positions (affine points) as the homogenized $\mathbb{R}^3$ points,* i.e. $\mathbb{R}^4$ *vectors with the last entry being* 1*. By simply having the last component setting to zero, we can represent object on the plane at infinity.*

In particular, we do not need to set huge numbers in their 3D coordinates.

This is particularly useful for representing a point light source which can be set at infinity. When the light source is at infinity, the light shed into the scene are parallel light rays. Parallel lights have just the information of a 3D directional vector, which are indeed those objects with the last entry 0.

8.2.5 Homography

A **collineation** is an invertible transformation on a projective space (with dimension ≥ 2) such that straight lines stay straight. To obtain linear perspective, any transformation of the geometric object must be a collineation.

Theorem 8.1 — Fundamental theorem of projective geometry. Every collineation on a projective space $\mathbb{RP}^n$ with dimension $n \geq 2$ is an invertible linear transformation (*i.e.* an invertible matrix) applied on the homogeneous coordinate $\mathbb{R}^{n+1}$.

Recall that affine transformations on $\mathbb{R}^3$:

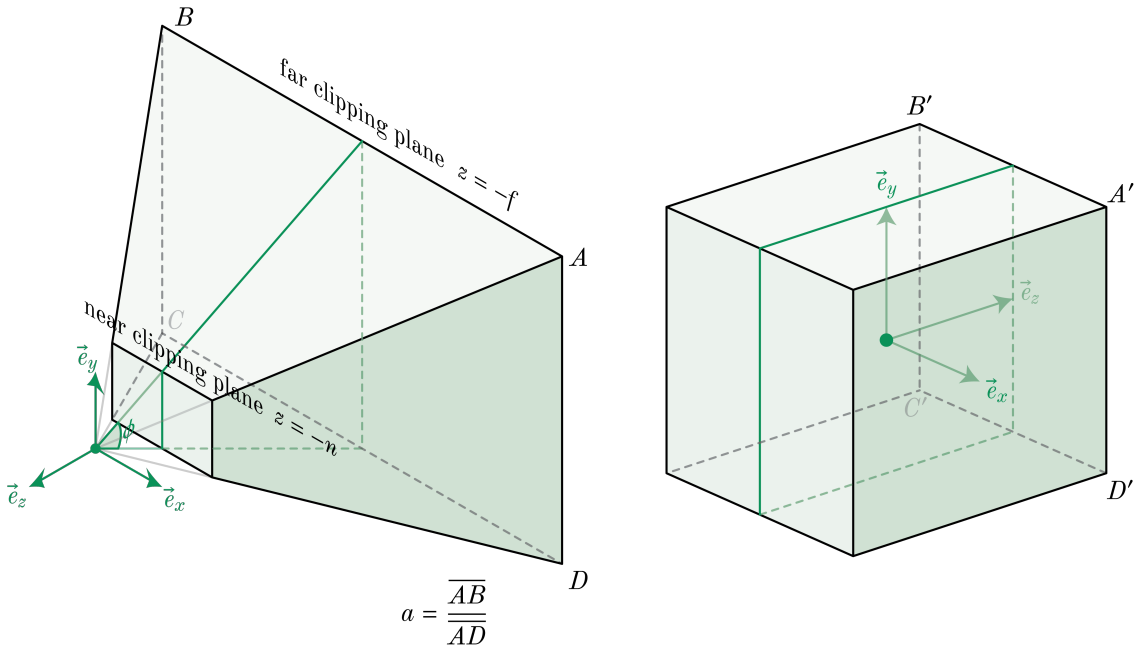
$$\text{(Affine transformation)} \quad \begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \tag{8.9}$$

are precisely special cases of these linear transformations (4×4 matrices) on the homogeneous coordinates ($\mathbb{R}^4$).

More generally, we can have matrices that does not have the last row being $0, 0, 0, 1$. The resulting transformation on the 3D space will no longer be affine, but will still preserves the notion of collinearity.

$$\begin{matrix} \text{(Collineation)} \\ \text{(Projective transformation)} \\ \text{(Homography)} \end{matrix} \quad \begin{bmatrix} p'_x/p'_w \\ p'_y/p'_w \\ p'_z/p'_w \\ 1 \end{bmatrix} \sim \begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ p'_w \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \tag{8.10}$$

These transformations on projective spaces induced from linear transformations on the homogeneous coordinates are called **homography** or **projective transformation**. (These term can be used for transformations on $\mathbb{RP}^1$, where collineation gives no information.)



(a) The viewing frustum in the camera coordinate.

(b) The viewing box in the normalized device coordinate.

Figure 8.5 The projection matrix is the collineation that maps the given viewing frustum in the camera coordinate to the viewing box in the normalized device coordinate.

8.3 Projective transformation into the normalized device coordinate

Suppose we have the position coordinates of vertices represented in the camera coordinate system. What we want to do is to map them into the normalized device coordinate (NDC). This map should (i) make objects further from the eye smaller in the NDC, and (ii) preserve collinearity. The second property suggests that this transformation must be a collineation. By the fundamental theorem of projective geometry, this transformation must be a 4×4 matrix applied to the 4-dimensional homogeneous coordinates of the positions.

To determine this 4×4 matrix, it turns out that it is enough to specify the desired region in the camara coordinate system that will be mapped to the viewing (clipping) box in the NDC. This region is called the **viewing frustum** (Figure 8.5).

The viewing frustum is a truncated rectangular cone. The apex of the cone is the eye. Hence, the viewing frustum expands larger as we move further from the eye. Therefore, as we map the frustum into the viewing box, the object that is further from the eye will be scaled smaller, creating the foreshortening phenomenon.

The viewing frustum, as illustrated in Figure 8.5, is described by four parameters:

- Field of view $\phi > 0$: the angle of view in the vertical direction.
- Aspect ratio $a > 0$: the $^{\text{width}}/_{\text{height}}$ of the base rectangle of the frustum.
- Distance $n > 0$ to the near clipping plane.
- Distance $f > 0$ ($f > n$) to the far clipping plane.

Task 8.2 Given the field of view angle ϕ, aspect ratio a, distance n,f to the near and far clipping planes, construct the 4×4 matrix $\mathbf{P}$ so that the viewing frustum is mapped to the viewing box as drawn in Figure 8.5.

That is, implement the function

$$\mathbb{R}^{4\times4} \ni \mathbf{P} = \text{Perspective}(\phi, a, n, f). \tag{8.11}$$

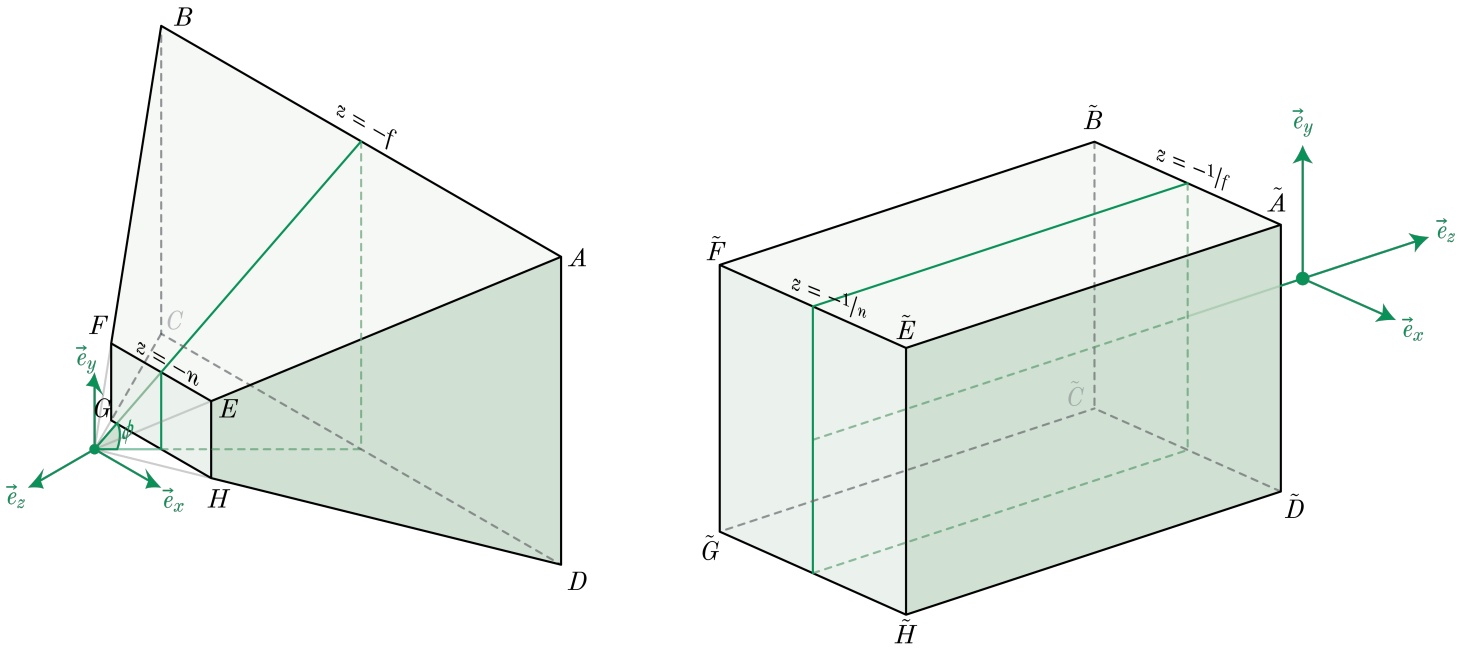
In the following, we derive the matrix $\mathbf{P}$ of Task 8.2 by a sequence of elementary moves.

8.3.1 Corners of the frustum

We should first work out the coordinates of the corner points of the frustum in Figure 8.6 (a). We know that the distance of the far clipping plane from the origin is f. Knowing the angle ϕ, we know that the y coordinate of both A and B is $f \cdot \tan(\phi/2)$. With the aspect ratio being a, we also know the x coordinates of A, B are $\pm af \tan(\phi/2)$. Similarly we have all coordinates of $A, \ldots, H$. In particular,

$$A = \begin{bmatrix} af\tan(\phi/2) \\ f\tan(\phi/2) \\ -f \\ 1 \end{bmatrix}, \quad E = \begin{bmatrix} an\tan(\phi/2) \\ n\tan(\phi/2) \\ -n \\ 1 \end{bmatrix}. \tag{8.12}$$

8.3.2 Sending the apex to infinity



(a) Before applying $\mathbf{R}$. **(b)** After applying $\mathbf{R}$.

Figure 8.6 The matrix $\mathbf{R} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -1 & \\ & & & -1 \end{bmatrix}$ sends the apex of the frustum to infinity, yielding a rectangular box.

The first step is to send the apex of the frustum (the origin, or the eye) to infinity in the z direction. By doing so, the sides of the frustum, specifically AE, DH, BF, CG, will become parallel to the z direction. This transformation will give us a rectangular box. See Figure 8.6.

In other words, we want a transformation that sends the origin $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^\mathsf{T}$ to the z-direction point at infinity $\begin{bmatrix}0 & 0 & \pm 1 & 0\end{bmatrix}^\mathsf{T}$. Also, we do not want to modify the already-established parallelism in the xy part. So, we use the matrix

$$\mathbf{R} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & -1 \\ & & -1 & \end{bmatrix} \tag{8.13}$$

which sends $\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^\mathsf{T}$ to $\begin{bmatrix}0 & 0 & -1 & 0\end{bmatrix}^\mathsf{T}$. We will see why we take the particular choice of the minus signs in a moment. For a general affine point, what $\mathbf{R}$ does is that it takes the reciprocal in the z component:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & -1 \\ & & -1 & \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ -1 \\ -p_z \end{bmatrix} \sim \begin{bmatrix} -p_x/p_z \\ -p_y/p_z \\ 1/p_z \\ 1 \end{bmatrix} \tag{8.14}$$

When $\mathbf{R}$ is applied to the corners of the frustum (8.12), we get

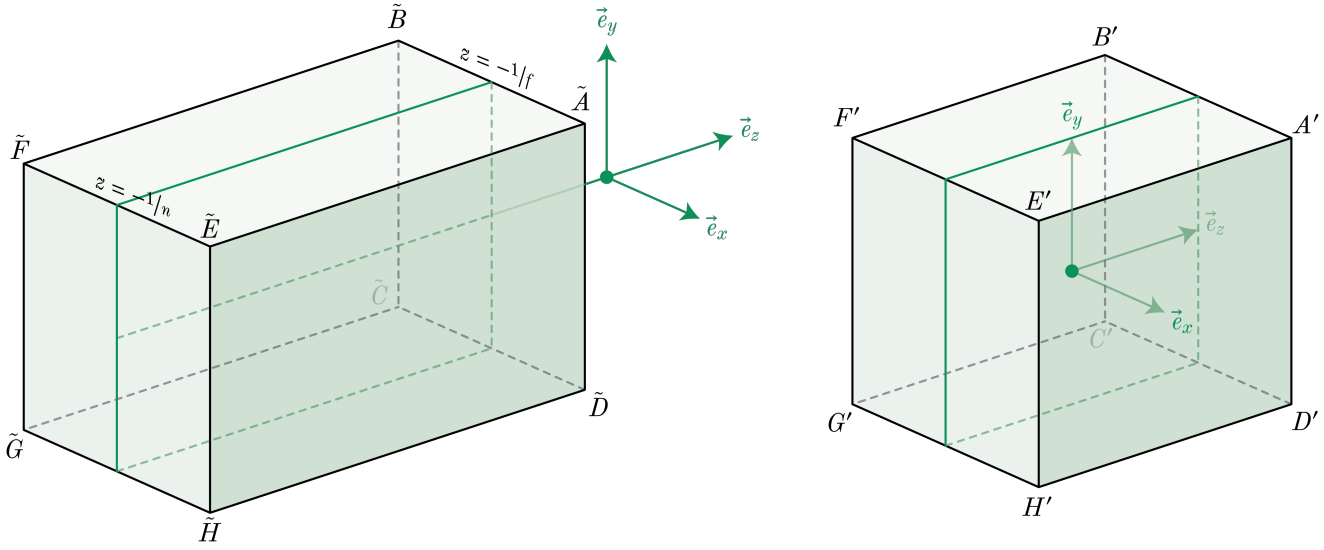
$$\tilde{A} = \mathbf{R}A \sim \begin{bmatrix} a\tan(\phi/2) \\ \tan(\phi/2) \\ -1/f \\ 1 \end{bmatrix}, \quad \tilde{E} = \mathbf{R}E \sim \begin{bmatrix} a\tan(\phi/2) \\ \tan(\phi/2) \\ -1/n \\ 1 \end{bmatrix}. \tag{8.15}$$

Note that letting $\mathbf{R} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$ (without the minus signs) will also undistort the frustum into a rectangular box, but then $\tilde{A} = \mathbf{R}A \sim \begin{bmatrix} -a\tan(\phi/2) \\ -\tan(\phi/2) \\ -1/f \\ 1 \end{bmatrix}$ has a negated xy coordinate. The sign choice that yields (8.15) is consistent with Figure 8.6 (b).

8.3.3 Translating and scaling into the viewing box

Now that Figure 8.6 (b) is a rectangular box, we just need to translate and scale it into the viewing box (Figure 8.7). The center of the box of Figure 8.6 (b) (equivalently Figure 8.7 (a)) is given by $\begin{bmatrix}0 & 0 & -\frac{1}{2}\left(\frac{1}{n}+\frac{1}{f}\right) & 1\end{bmatrix}^\mathsf{T}$. The half-sidelengths of the box is $a\tan(\phi/2)$ in the x direction, $\tan(\phi/2)$ in the y direction, and $\frac{1}{2}\left(\frac{1}{n}-\frac{1}{f}\right)$ in the z direction. Therefore, we first translate the box so that it is centered at the origin, using the translation matrix

$$\mathbf{T} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & \frac{1}{2}(1/n + 1/f) \\ & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & \frac{f+n}{2fn} \\ & & & 1 \end{bmatrix}. \tag{8.16}$$



(a) Before applying **ST**. **(b)** After applying **ST**.

Figure 8.7 A translation $\mathbf{T} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & \frac{1}{2}(1/n+1/f) \\ & & & 1 \end{bmatrix}$ followed by a scaling $\mathbf{S} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \left(\frac{1}{2}(1/n-1/f)\right)^{-1} & \\ & & & 1 \end{bmatrix}$ sends the rectangular box of Figure 8.6 (b) to the viewing box $\mathbb{V}_{3D} = [0,1]^3$.

Then we scale the box so that the half-sidelengths are all 1:

$$\mathbf{S} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \frac{1}{\frac{1}{2}(1/n-1/f)} & \\ & & & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \frac{2fn}{f-n} & \\ & & & 1 \end{bmatrix}. \tag{8.17}$$

The two elementary operations combine into

$$\mathbf{ST} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \frac{2fn}{f-n} & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \frac{f+n}{2fn} \\ & & & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \frac{2fn}{f-n} & \frac{f+n}{f-n} \\ & & & 1 \end{bmatrix}. \tag{8.18}$$

One can take (8.15) and check that indeed

$$A' = \mathbf{ST}\tilde{A} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad E' = \mathbf{ST}\tilde{E} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \tag{8.19}$$

8.3.4 Final formula

Combining all moves (8.13) and (8.18), our perspective projection matrix is given by

$$\mathbf{P} = \mathbf{STR} = \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & \frac{2fn}{f-n} & \frac{f+n}{f-n} \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & -1 \\ & & -1 & \end{bmatrix} \tag{8.20}$$

$$= \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ & & -1 & \end{bmatrix} \tag{8.21}$$

Summary 8.1 Given the field of view angle ϕ, aspect ratio a, distance n,f to the near and far clipping planes, the 4×4 matrix $\mathbf{P}$ that maps the viewing frustum to the viewing box as shown in Figure 8.5 is given by

$$\mathbb{R}^{4\times 4} \ni \mathbf{P} = \text{Perspective}(\phi, a, n, f) \tag{8.22}$$

$$= \begin{bmatrix} \frac{1}{a\tan(\phi/2)} & & & \\ & \frac{1}{\tan(\phi/2)} & & \\ & & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ & & -1 & \end{bmatrix}. \tag{8.23}$$

Exercise 8.1 Complete the implementation of the Camera class. The Camera class has additional members (`fovy`, `aspect`, `near`, `far`, and their default values) that correspond to the arguments of (8.22). Specifically, the method `void Camera::computeMatrices(void)` should update the projection matrix `proj`. Note that GLM matrix element access is column major.

Code 8.1 `Camera.h`

```
/*************************************************
Camera is a class for a camera object.
*************************************************/
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>

#ifndef __CAMERA_H__
#define __CAMERA_H__

class Camera {
public:
    glm::vec3 eye;// position of the eye
    glm::vec3 target;  // look at target
    glm::vec3 up;      // up vector
    float fovy;  // field of view in degrees
    float aspect; // aspect ratio
    float near; // near clipping distance
```

```
18      float far; // far clipping distance
19
20      // default values for reset
21      glm::vec3 eye_default = glm::vec3(5.0f, 0.0f, 0.0f);
22      glm::vec3 target_default = glm::vec3(0.0f, 0.0f, 0.0f);
23      glm::vec3 up_default = glm::vec3(0.0f, 1.0f, 0.0f);
24      float fovy_default = 30.0f;
25      float aspect_default = 4.0f/3.0f;
26      float near_default = 0.01f;
27      float far_default = 100.0f;
28
29      glm::mat4 view = glm::mat4(1.0f);    // view matrix
30      glm::mat4 proj = glm::mat4(1.0f);    // projection matrix
31
32      void rotateRight(const float degrees);
33      void rotateUp(const float degrees);
34      void computeMatrices(void);
35      void reset(void);
36  };
37
38  #endif
```

9. Hierarchical Modeling

Hierarchical modeling is the technique of organizing the geometric objects in a scene into a **scene graph**. A scene graph is a data structure that represents a network of belonging relationships. This hierarchical data structure is important for:

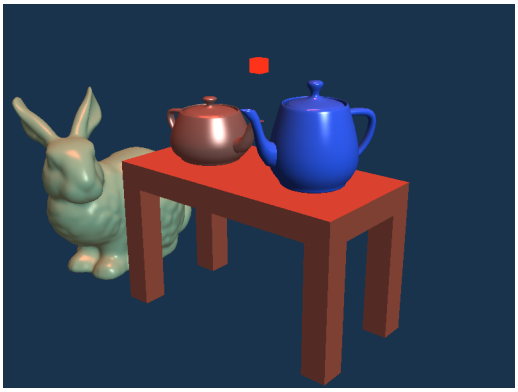
- Organizing complex scene into semantically meaningful groups; *e.g.* table legs and table top belong to a group called table.
- Rigging an animated character; *e.g.* fingers are connected to a hand, the hand is connected to forearm, the forearm is connected to the upper arm, *etc*. Moving the forearm should move all the descendents (*i.e.* the forearm, the hand and the fingers) together as a group.
- Fast algorithms for geometry query. Geometries can be grouped into boxes, and nearby boxes can be grouped together as a bigger box, *etc*. Examples of such spatial data structures include *octree*, *k-D tree*, and *bounding volume hierarchy*. Geometry query problems, such as finding the nearest geometry of a given point or finding the geometry hit by a given light ray, can be simplified into query problems within only the relevant boxes.

9.1 Scene description

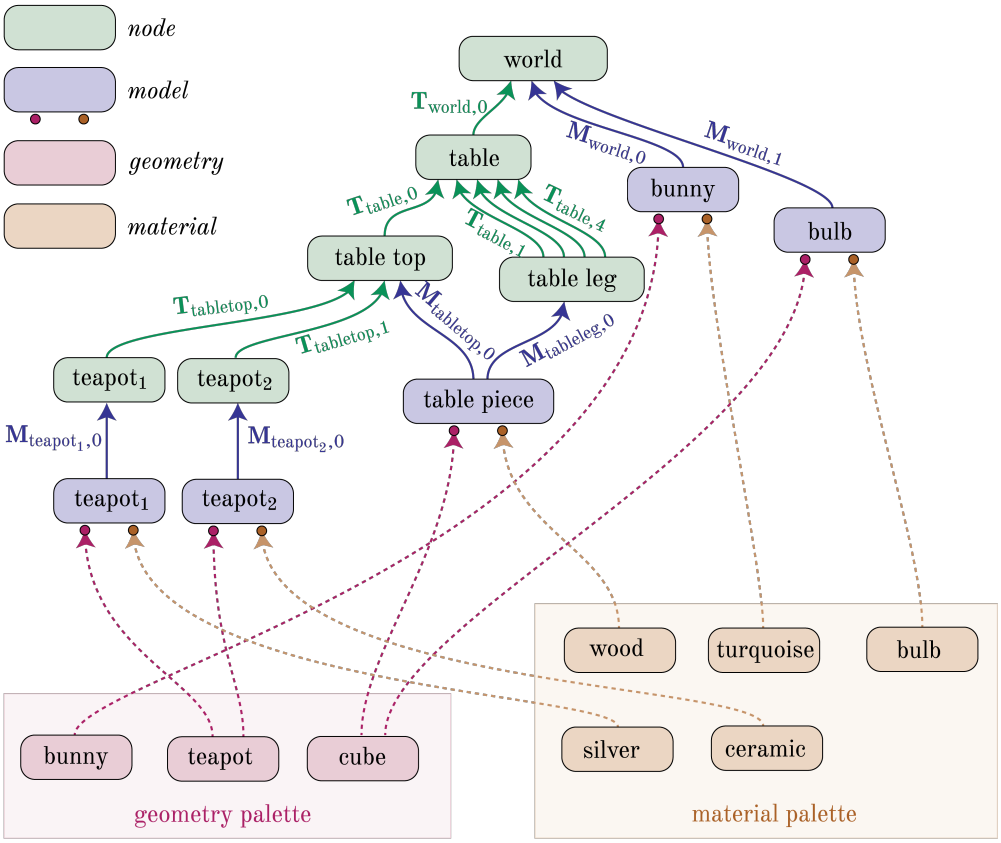
An example of a scene diagram is illustrated in Figure 9.1. Let us first talk about the lower portion of the diagram involving models, geometry and material palettes.

9.1.1 Memory-efficient modeling

In such a scene, we want to draw geometric objects, namely a bunny, a table made of several transformed cubes, two teapots and a small cube suspended above the table representing a light bulb geometry. Note that there are repeated geometries such as teapots. They are drawn with different shading parameters.



(a) Rendered scene.



(b) Scene graph.

Figure 9.1 A scene graph is a data structure capturing the hierarchical and transformational relationship among semantically meaningful groups.

Instead of duplicating teapot's entire set of buffers for drawing two slightly different-looking teapots, we introduce a notion of **model**.

Definition 9.1 A **model** is a geometry with material.

Here, a material is just a set of uniform variables that are needed to tune the shader. We will learn what each of the material parameters mean when we talk about lighting.

```
struct Material {
    glm::vec4 ambient = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
    glm::vec4 diffuse = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);
    glm::vec4 specular = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
    glm::vec4 emision = glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
    float shininess = 10.0f;
};
struct Model {
    Geometry* geometry;
    Material* material;
};
```

Note that each model is just a pointer to a geometry and a pointer to a material. As shown in Figure 9.1, there are two teapot models, but we only need to have one teapot geometry. In particular, the memory it costs only amount to one teapot geometry, and a few bytes for the materials and the pointers. This strategy is much more memory-efficient than storing two teapot geometries.

Summary 9.1 In a scene, we prepare a geometry palette and a material palette that store the source geometries and the source materials. The object that we draw are in the form of models. A model has a pointer to a geometry and a pointer to a material in the palettes. When we draw a model, we set the shader parameters according to the material, and we draw the geometry pointed to by the model.

9.1.2 Reading a scene graph

The models that we want to draw are organized into a graph. The graph consists of nodes (green "generic" node and blue "model" node) and connections with direction. Every node has its own coordinate system. Note that the coordinate systems are *not* explicitly given by a set of basis vectors and a choice of origin relative to the world. Rather, the coordinate systems are implicitly represented by the 4-by-4 transformation matrices equipped on each connection, recording the relative change of coordinates between two connected nodes.

▪ **Example 9.1** In Figure 9.1, suppose we want to draw the table piece model of the table top of the table. Then the model matrix is given by

$$\mathbf{T}_{\text{world},0}\mathbf{T}_{\text{table},0}\mathbf{M}_{\text{tabletop},0} \tag{9.1}$$

If we have a camera with view matrix $\mathbf{V}$. Then the modelview matrix is set to be

$$\mathbf{V}\mathbf{T}_{\text{world},0}\mathbf{T}_{\text{table},0}\mathbf{M}_{\text{tabletop},0} \tag{9.2}$$

as we draw the cube geometry. ▪

Although there are only 5 model nodes, we will draw 9 models. There are in total 9 paths that go from a model node to the world node. In particular, the "table piece" model is drawn 5 times.

9.1.3 Data structure

While drawing the diagram of Figure 9.1 (b) is intuitive on a piece of paper, we need to encode it into a practical data structure.

The data structure consists of a container of **nodes** (the green nodes in Figure 9.1 (b)), and a container of **models**.

Each node can be wired from below to a node or to a model. This wiring information is stored as member variables in each node. Specifically, each node is equipped with an array of pointers to child nodes, an array of transformation matrices associated with the connections to those child nodes, an array of pointers to the models, and the transformation matrices associated with the connections to those models.

```
struct Node {
    std::vector< Node* > childnodes;
    std::vector< glm::mat4 > childtransforms;
    std::vector< Model* > models;
    std::vector< glm::mat4 > modeltransforms;
};
```

Now, define a container of nodes, a container of models, and the containers for the geometry and material palettes. We use `std::map` key-value pairs for the containers.

```
std::map< std::string, Node* > node;
std::map< std::string, Model* > model;
std::map< std::string, Geometry* > geometry;
std::map< std::string, Material* > material;
```

Then, Figure 9.1 (b) is equivalent to the following script. We assume the geometry and material palettes have already been setup. That is, the pointer `geometry["teapot"]`$\in$ `Geometry*` points to a geometry with properly setup vertex array object. First, we can create the set of nodes and models.

```
// Set of nodes
node["world"] = new Node;
node["table"] = new Node;
node["table top"] = new Node;
node["table leg"] = new Node;
node["teapot1"] = new Node;
node["teapot2"] = new Node;
// Set of models
model["teapot1"] = new Model;
model["teapot1"] -> geometry = geometry["teapot"];
model["teapot1"] -> material = material["silver"];
model["teapot2"] = new Model;
model["teapot2"] -> geometry = geometry["teapot"];
model["teapot2"] -> material = material["ceramic"];
model["table piece"] = new Model;
model["table piece"] -> geometry = geometry["cube"];
model["table piece"] -> material = material["wood"];
model["bunny"] = new Model;
model["bunny"] -> geometry = geometry["bunny"];
```

```
20: model["bunny"] -> material = material["turquoise"];
21: model["bulb"] = new Model;
22: model["bulb"] -> geometry = geometry["cube"];
23: model["bulb"] -> material = material["bulb"];
```

Now, the containers of nodes and models are filled. But we have not specified their connection. In the following, we wire the connections between the nodes by appending values into the `std::vector<>` members in each node.

```
24: // Set connection
25: node["table"] -> childnodes.push_back( node["table top"] );
26: node["table"] -> childtransforms.push_back( T_table,0 );
27: node["table"] -> childnodes.push_back( node["table leg"] );
28: node["table"] -> childtransforms.push_back( T_table,1 );
29: node["table"] -> childnodes.push_back( node["table leg"] );
30: node["table"] -> childtransforms.push_back( T_table,2 );
31: node["table"] -> childnodes.push_back( node["table leg"] );
32: node["table"] -> childtransforms.push_back( T_table,3 );
33: node["table"] -> childnodes.push_back( node["table leg"] );
34: node["table"] -> childtransforms.push_back( T_table,4 );
35: node["table leg"] -> models.push_back( model["table piece"] );
36: node["table leg"] -> modeltransforms.push_back( M_tableleg,0 );
37: node["table top"] -> models.push_back( model["table piece"] );
38: node["table top"] -> modeltransforms.push_back( M_tabletop,0 );
39: node["table top"] -> childnodes.push_back( node["teapot1"] );
40: node["table top"] -> childtransforms.push_back( T_tabletop,0 );
41: node["table top"] -> childnodes.push_back( node["teapot2"] );
42: node["table top"] -> childtransforms.push_back( T_tabletop,1 );
43: node["teapot1"] -> models.push_back( model["teapot1"] );
44: node["teapot1"] -> modeltransforms.push_back( M_teapot1,0 );
45: node["teapot2"] -> models.push_back( model["teapot2"] );
46: node["teapot2"] -> modeltransforms.push_back( T_teapot2,0 ); );
47: node["world"] -> childnodes.push_back( node["table"] );
48: node["world"] -> childtransforms.push_back( T_world,0 );
49: node["world"] -> models.push_back( node["bunny"] );
50: node["world"] -> modeltransforms.push_back( M_world,0 );
51: node["world"] -> models.push_back( model["bulb"] );
52: node["world"] -> modeltransforms.push_back( M_world,1 );
```

Just as above, every solid arrow in Figure 9.1 (b) amounts to two lines of code in this script.

The next task we have is the following.

Task 9.1 Given a scene graph as illustrated in Figure 9.1 (b) and represented by a data structure as described in this section, construct an algorithm that traverses over every path that go from a model to the world node, and subsequently draw the geometry with the correct modelview matrix in an efficient manner.

An algorithm will become apparent after some abstraction.

9.2 Formal properties of a scene graph

The scene graph is a directed graph with some additional conditions. First of all, let us define a directed graph.

9.2.1 Directed graphs

A directed graph is a set of nodes with a set arrows joining a few pairs of the nodes.

Definition 9.2 — Directed graph. A **directed graph** is a structure $(\mathscr{N}, \mathscr{E}, \mathrm{src}, \mathrm{dst})$ consisting of

- a finite set of nodes $\mathscr{N}$,
- a finite set of directed edges $\mathscr{E}$,
- a function $\mathrm{src}\colon \mathscr{E} \to \mathscr{N}$ returning the source node of a given edge, and
- a function $\mathrm{dst}\colon \mathscr{E} \to \mathscr{N}$ returning the destination node of a given edge.

Note that there can be multiple edges sharing the same source and same destination. (For example, the multiple edges between "table leg" and "table" in Figure 9.1 (b).)

In a scene graph such as Figure 9.1 (b), the "world" node is a **sink** (analogous to the root of a tree) as arrows only point in but there is no arrow out from this sink. A "model node" is a **source** (analogous to a leaf of a tree) as an adjacent arrow only goes out.

Definition 9.3 — Sink and source. For a directed graph $(\mathscr{N}, \mathscr{E}, \mathrm{src}, \mathrm{dst})$, a node $a \in \mathscr{N}$ is called a **sink** if there is no $e \in \mathscr{E}$ such that $\mathrm{src}(e) = a$. A node $b \in \mathscr{N}$ is called a **source** if there is no $e \in \mathscr{E}$ such that $\mathrm{dst}(e) = b$.

We say that a node a is reachable to node b if there is a path connecting from a to b, formalized as follows.

Definition 9.4 — Reachability relation. Every directed graph $(\mathscr{N}, \mathscr{E}, \mathrm{src}, \mathrm{dst})$ gives rise to a so-called **reachability relation** $\preceq\colon \mathscr{N} \times \mathscr{N} \to \{\mathrm{True}, \mathrm{False}\}$, defined by the following rules:

- $a \preceq a$ is true for all $a \in \mathscr{N}$;
- $\mathrm{src}(e) \preceq \mathrm{dst}(e)$ is true for all $e \in \mathscr{E}$;
- if $a \preceq b$ and $b \preceq c$ are true, then $a \preceq c$ is true.

An important condition for our scene graph is that, if we follow the arrows, we should not run into a closed loop. This type of directed graphs are called a **directed acyclic graph** (**DAG**).

Definition 9.5 — DAG. A directed graph $(\mathscr{N}, \mathscr{E}, \mathrm{src}, \mathrm{dst})$ is called a **directed acyclic graph** (**DAG**) if the reachability relation $\preceq$ of the directed graph satisfies the property that

$$\text{if } a \preceq b \text{ and } b \preceq a, \text{ then } a = b. \tag{9.3}$$

(Equivalently, $(\mathscr{N}, \preceq)$ is a partial ordered set.)

Now, the conditions for a scene graph are the following. (i) A scene graph must

be a DAG. (ii) A scene graph must also have a single sink called "world."

Definition 9.6 — Rooted DAG. We call a DAG a **rooted DAG** if it only has a single sink.[a]

[a] Equivalently, the partial ordered set $(\mathcal{N}, \preceq)$ is an **upper-semilattice**.

Summary 9.2 A scene graph is a rooted DAG. The source nodes are the models, and the single sink is the world node.

Note that such a graph is more general than a **tree**. A tree should not have any undirected cycle. A scene graph can have undirected cycles. In Figure 9.1, there are several paths going from "table piece" to "world." For a tree, any two nodes admit at most one path.

9.2.2 Stack

Let us review a data structure called **stack**. Stack is required for solving Task 9.1.

A stack is a data structure for temporarily store objects waiting to be processed.

Let $\mathbb{T}$ be a generic data type. A stack $\mathbf{a}$ of type $\mathbb{T}$ is an ordered list of objects in $\mathbb{T}$ with an arbitrary length

$$\mathbf{a} = (a_1, \dots, a_k) \in \bigcup_{n=0}^{\infty} \mathbb{T}^n, \quad a_i \in \mathbb{T}. \tag{9.4}$$

Note that it is possible that a stack has size zero, in which case we call the stack empty $() \in \mathbb{T}^0$. The following operations are available for navigating in the space of stacks:

- **push**: $\mathbb{T}^k \times \mathbb{T} \to \mathbb{T}^{k+1}$; $(a_1, \dots, a_k)$.**push**$(b) := (a_1, \dots, a_k, b)$.
- **pop**: $\mathbb{T}^k \to \mathbb{T}^{k-1}$; $(a_1, \dots, a_{k-1}, a_k)$.**pop**$() := (a_1, \dots, a_{k-1})$.

There is also a function that helps us read the last entry of the stack **top**: $\mathbb{T}^k \to \mathbb{T}$; $(a_1, \dots, a_k)$.**top**$() := a_k$.

In particular, the stack operations only manipulate the end of the list. In contrast to a **queue**, whose operating priority is first in first out, the stack operating priority is **first in last out**.

9.2.3 Traversing over a rooted directed acyclic graph

Let us solve Task 9.1 for a general rooted DAG.

Task 9.2 Given a rooted DAG $(\mathcal{N}, \mathcal{E}, \text{src}, \text{dst})$, find all paths connecting a source to the single sink.

The algorithm to do so is the **depth-first search** (**DFS**).

Similar to the data structure explained in Section 9.1.3, we consider for each node $a \in \mathcal{N}$ the set of connecting edges that point into a.

Definition 9.7 For each $a \in \mathscr{N}$, let the set of child node connections be denoted by

$$\mathrm{child}(a) = \mathrm{dst}^{-1}(\{a\}) = \{e \in \mathscr{E} \mid \mathrm{dst}(e) = a\} \subset \mathscr{E}. \tag{9.5}$$

Now, let $(\mathscr{N}, \mathscr{E}, \mathrm{src}, \mathrm{dst})$ be a rooted DAG with $a_0 \in \mathscr{N}$ be the unique sink.

Algorithm 9.1 Depth-first search for a rooted DAG

```
Let S = ( ) be an empty stack.
Let a = a_0 be the current node.
S.push(a).
If we have reached a source, then we have explored a path.
while S is not empty do
    a ← S.top();   S.pop();
    for e ∈ child(a) do
        S.push(src(e));
    end for
end while
```

Note 9.1 *Note that in a traditional depth-first search, we need to label visited nodes as "discovered;" if an element top-popped out from the stack has already been visited, we would have to discard that element and top-pop the next entry from the stack. However, that is not what we want. We want to visit all possible paths joining a_0 to the bottom of the graph. An intermediate node can be visited multiple times.*

If the graph is not a DAG, then this depth-first search will run into an infinite loop.

One can detect whether the algorithm has run into an infinite loop by checking the size of the stack. When the graph is not a DAG and has a directed cycle, then the stack will grow indefinitely. In contrast, for a DAG, the size of the stack will never be more than the total number of nodes in the graph.

Therefore, one may throw an error message when the stack size is greater than the total number of nodes.

9.3 Draw the scene

Suppose we have a scene graph as described in Section 9.1.3. We can now traverse over all paths connecting the "world" node and each model.

If we directly translate Algorithm 9.1 to our context, we have

Algorithm 9.2 Depth-first search for the scene graph

```
1: std::stack< Node* > NodeStack;
2: Node* current_node = node["world"];
3: NodeStack.push( current_node );
4: while NodeStack is nonempty do
5:     current_node = NodeStack.top();  NodeStack.pop();
6:     // Insert here: run through the models (current_node -> models) and draw
   each of them.
7:     for childnode ∈ (current_node -> childnodes) do
8:         NodeStack.push( childnode );
9:     end for
10: end while
```

Indeed, Algorithm 9.3 will visit all possible paths joining the world and each model. However, when we want to draw the model, we find ourselves missing an important information. We do not have the modelview matrix handy.

To access the modelview matrix of the current path, we maintain another stack alongside the `NodeStack`. This second stack is the so-called **matrix stack**.

9.3.1 Matrix stack

The matrix stack is a stack of 4-by-4 matrices.

```
std::stack< glm::mat4 > MatrixStack;
```

At all time, `MatrixStack` and `NodeStack` have the same size. Moreover, at all time, for `NodeStack`$= (a_1, \ldots, a_k)$ and `MatrixStack`$= (\mathbf{A}_1, \ldots, \mathbf{A}_k)$, we have that

$$\mathbf{A}_i \text{ is the modelview matrix for node } a_i \tag{9.6}$$

for all $i = 1, \ldots, k$.

By maintaining the matrix stack alongside the node stack, we have the modelview matrix state consistent with the state of the current node.

Let us see how to let the condition (9.6) hold for all time. All we ever do to modify `NodeStack` is pushing and popping the stack. When we simultaneously pop the stacks `NodeStack` and `MatrixStack`, we obviously have (9.6) stay true. So, the only thing we need to worry about is pushing an element into the stack.

The only time we push a node into the `NodeStack` in Algorithm 9.3 is at Line 8 (and the initialization at Line 3). When we push a child-node into the stack, the corresponding new matrix that we should push into the matrix stack is $\mathbf{AT}$, where $\mathbf{A}$ is the modelview matrix for the `current_node`, and $\mathbf{T}$ is the transformation matrix on that child-node connection.

With that, (9.6) stays true for all time.

> **Summary 9.3** In conclusion, the draw procedure for the scene graph is given by the following pseudocode.

Algorithm 9.3 Depth-first search with a matrix stack

```
1: std::stack< Node* >     NodeStack;
   std::stack< glm::mat4 > MatrixStack;
2: Node*     current_node   = node["world"];
   glm::mat4 current_matrix = the view matrix of the camera;
3: NodeStack.push  ( current_node   );
   MatrixStack.push( current_matrix );
4: while NodeStack is nonempty do
5:    current_node = NodeStack.top();  NodeStack.pop();
    current_matrix = MatrixStack.top();  MatrixStack.pop();
6:    for i = 0,...,(current_node -> models).size() - 1 do   ▷ draw each model
7:       model = current_node -> models[i];
          M =current_node -> modeltransforms[i];
8:       Set shader's modelview matrix as (current_matrix * M).
9:       Set shader's material parameters according to (model -> material).
10:      model -> geometry -> draw();
11:   end for
12:   for j = 0,...,(current_node -> childenodes).size() - 1 do
13:      childnode = current_node -> childnodes[j];
          T =current_node -> childtransforms[j];
14:      NodeStack.push( childnode );
         MatrixStack.push( current_matrix * T );
15:   end for
16: end while
```

III Lighting and Textures

10. Lighting

Music is the arithmetic of sounds
as optics is the geometry of light.

Claude Debussy, 1862–1918

In rendering, lighting is the process of determining the color based on an approximation of optical physics. In a physically realistic lighting, we need to simulate how the lights propagate and interact with the scene geometries. There are many paradigms for simulating optical physics, ranging from physically accurate but complicated models to approximations but efficient models:

- Electromagnetic equations (wave optics). Lights are electromagnetic waves governed by Maxwell's equations.
- Fresnel equations. It is a special case of electromagnetic waves where waves are plane waves moving in straight lines. That is, lights are polarized rays. They reflect and refract off a surface into mutated polarized rays by known formulae.
- Geometric optics (ray optics). Lights are (densities of) rays without the wave nature. Rays travel in straight lines until they are blocked by a surface, in which case they reflect and refract off a surface by Snell's law or by scattering models.

The wave nature of light comes into play if the lights are interacting with geometric features that vary at a scale comparable to the wavelength of the light. Wave optics phenomena include dispersions (different wavelengths travel in different speed, causing separations of colors through a prism), diffraction (lights craw over an edge of a wall), interference (pattern on a soap film or a film of oil on water), *etc*. In most applications in rendering, geometric optics are sufficiently accurate.

For geometric optics, one still need to solve a difficult **global illumination** problem.

That is, there are infinitely many light rays, counted in densities (power per area per orientation), interweaving the space between geometries in a scene. The methods to evaluate this light field include **computational radiosity** and **ray tracing**.

10.1 Direct lighting in rasterization-based graphics

In a rasterization-based graphics pipeline, we do not have much resources to compute the global illumination.

- Before rasterization, the shader only sees the local geometry (per-vertex computation).
- Rasterization can be thought of as a single pass of determining how the light rays per pixel intersect the geometries.
- After rasterization, we only have access to fragments, and this is already the final stage where we need to determine the color. There is no access to other geometries in the fragment shader.

In particular, we only see how the light rays of the camera interact with the geometry, but we have no opportunity to compute a secondary interaction of a reflected ray with other geometries. Theoretically, we could feed the entire scene data into the shader (by a huge list of uniform variables), but this is not practical.

Later, we will learn that we could feed an image data into the shader as textures. However, these image data are 2D and it is not clear how to arrange the required 3D information for optical computations into a single 2D image. (There is one technique that does it, which is the method of shadow casting.)

Therefore, in rasterization-based lighting, we will simply ignore the secondary light-geometry interactions. Instead of feeding the entire scene into the shader, we will only feed a few light source into the shader. The color computation is only a function of the relative positions and orientations between the camera, the lights, and the local geometry (vertex or fragment) being shaded. This is called a **direct lighting**.

10.1.1 Gouraud shading and Phong shading

We can compute the lighting color in the vertex shader, and then let the color get linearly interpolated across each triangle during rasterization. This is called **Gouraud shading**.

Another way is to compute the lighting color in the fragment shader. It is the normal vector (orientation of the surface) that is interpolated during the rasterization. This scheme is called the **Phong shading**.

As shown in Figure 10.1, Phong shading shows much less discretization artifact when we have a low resolution mesh.

The color from lighting can change vary rapidly over the geometry (*e.g.* the highlight on a smooth surface). In particular, the amount of details in the lighting result can be at a higher resolution than the mesh geometry. So, interpolating the lighting results, which down samples the color information to the mesh resolution, give us a lower resolution quality than what we expect. In contrast, the parameters that are required to compute the lighting are often just the geometric information, such as the normal vectors; these parameters have the same resolution as the mesh

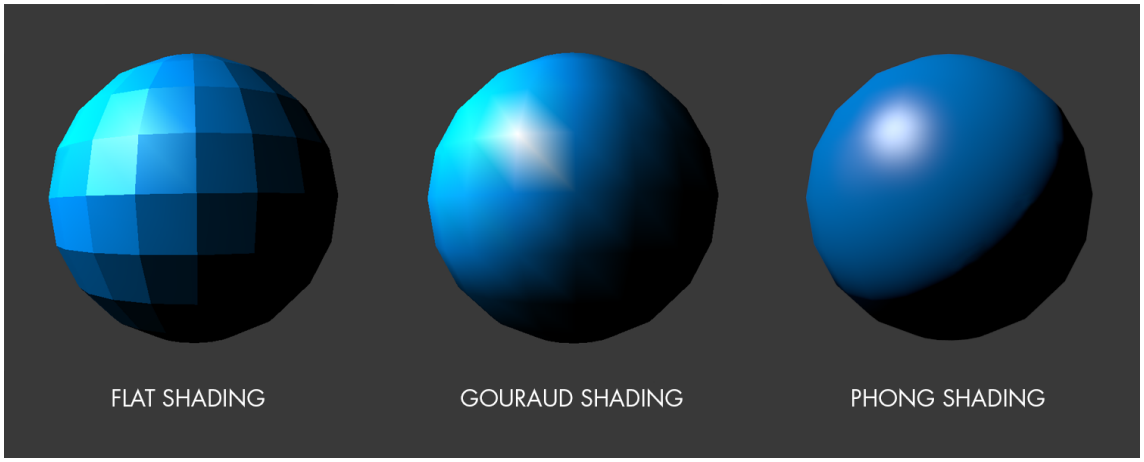


Figure 10.1 Flat shading computes one color per face. Gouraud shading computes colors per vertex, and then the colors are interpolated per face. Phong shading computes the colors per fragment (pixel) where all the parameters that determine the colors are interpolated.

geometry. Therefore, it is more advantageous to interpolate the normals (parameters) for lighting, because this interpolation does not loose information. That is, the interpolated normal vector is a rather faithful representation of the normal vector across the triangle. Computing the color per pixel from a faithfully represented set of parameters give us an accurate result at the pixel resolution.

We will use the Phong shading model. That is, the colors are computed per fragment.

10.2 Color vector

Let $\mathbf{L} = (L_R, L_G, L_B) \in \mathbb{R}^3$ represent a colored light. The components represent the light intensity in the red, green and the blue channel.[1] Each light channel interacts with geometries independently; in other words, a red light won't mutate into a green light after a reflection.[2]

A colored light can be modified by a vector-vector componentwise multiplication. Let $\mathbf{C} = (C_R, C_G, C_B) \in \mathbb{R}^3$. Then

$$\mathbf{CL} = (C_R, C_G, C_B)(L_R, L_G, L_B) := (C_R L_R, C_G L_G, C_B L_B) \tag{10.1}$$

is a possible mutation of a colored light. The typical example is that $\mathbf{C} = (C_R, C_G, C_B) \in \mathbb{R}^3$ describes the *color of a material*. If $\mathbf{L}$ is the light shined on the material, then $\mathbf{CL}$ represents the light reflected from the material. "$\mathbf{C} = (1, 0.5, 0)$" means that the material absorbs all blue lights, reflects all red lights, and reflects a bit of green lights. A white light $\mathbf{L} = (a, a, a)$, where $a \in \mathbb{R}$ is the intensity, will be reflected into some light with an orange tint $\mathbf{CL} = (a, \frac{a}{2}, 0)$.

[1] In the physical reality, a colored light is an infinite dimensional vector, with one intensity per infinitely possible wavelengths. We will learn about the correspondence between the physical colored light and the RGB 3D vector when we talk about color science. Unless we need to simulate full-spectrum dispersion, the RGB representation is good enough.

[2] Doppler's effect is one exception where the reflected/observed frequency of the light can shift.

Summary 10.1 Both colored light **L** and material color **C** are RGB vectors, but they have different meanings. The **C** vectors typically have component ranging between 0 and 1, representing how much light should pass after interacting with the material. On the other hand, **L** can have values greater than one.

C is unit-less. **L** has a physical unit of power (energy per unit time), or power per unit area, or power per unit area and angle, depending on which geometry the vector is assigned to.

The distinction between **C**-type vectors and **L**-type vectors are like the multiplicative version of vectors and positions.

- **C**-type color times a **L**-type color is an **L**-type vector.
- **C**-type color times a **C**-type color is a **C**-type vector.
- **L**-type color times a **L**-type color doesn't physical sense.

10.2.1 High dynamic range

The final pixel color is a **L**-type color. It is the total power of lights that intersects with the eye position of the camera through the pixel. In particular, it is possible that some channel of **L** has a value that is greater than 1. In that case, the color shown on the screen is the value clamped down to 1. This is called the light is **saturated** or **out of range**.

For example, $\mathbf{L} = (0.1a, 0.1a, a)$ is proportional to $(0.1, 0.1, 1)$, which should be blue. But if a too big, say $a = 100$, then $\mathbf{L} = (10, 10, 100)$ will only be shown as $\text{clamp}(\mathbf{L}) = (1, 1, 1)$, which is white. The image will appear to be overly exposed.

A **high dynamic range** (HDR) image is to store **L** per pixel as the unclamped floating point number, rather than a traditional $\{0, 1, \ldots, 255\}$ 8-bit integer that only represents a fractional number within $[0, 1]$. When a high dynamic range image is shown on a device supporting only 8-bit integer, or that the image needs to be stored in an 8-bit integer format, a **tone mapping** is applied.

Here is an example of tone mapping. Suppose $L_{\text{original}} > 0$ is a real number possibly reaching far beyond $[0, 1]$. We modify it into L_{mapped} by

$$L_{\text{mapped}} = AL_{\text{original}}^{\gamma}. \tag{10.2}$$

Typically $A = 1$. When $0 < \gamma < 1$, we squeeze the range of L_{original} so that L_{mapped} fits better into $[0, 1]$. We can also take an L_{mapped} and take a power of $\gamma > 1$ to map it back to the original physical intensity. This example of tone mapping is called **gamma correction**.

10.3 Reflection model

Computing the reflected light of a fragment is more than just multiplying a material color **C** with the in-coming light **L**. The reflected color is also a function of the geometric relationship among the viewing direction, light direction and the orientation of the surface.

A reflection model is a mathematical formula that returns the fragment color given

- the view direction **v** (the unit vector pointing from the geometry to the viewer),

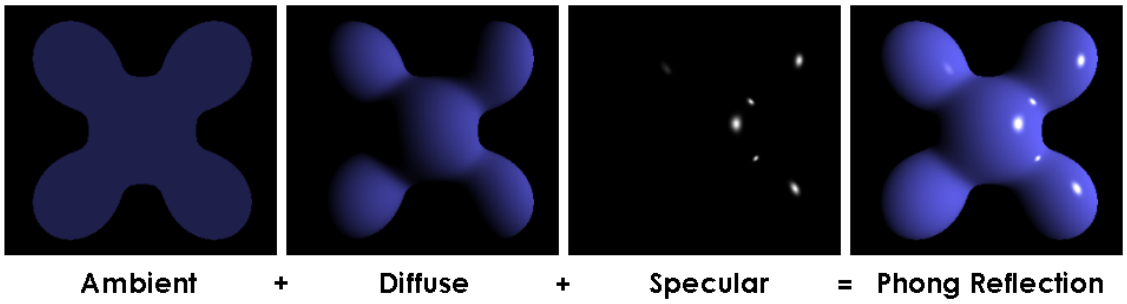


Figure 10.2 Phong reflection model.

- the light direction(s) $\mathbf{l}$ (the unit vector(s) pointing from the geometry to the light source(s)),
- the distance(s) d from the geometry to the light source(s)
- the normal direction $\mathbf{n}$ of the geometry.

These vectors are represented in the same coordinate system, and this coordinate system is isometric to the physical world coordinate system because we will use its Euclidean geometric structure. For example, the camera coordinate or the world coordinate are fine to use; for non-example, the normalized divice coordinate or the model coordinate may have a non-isometric stretching relative to the world.

The **Phong reflection model** is the common paradigm in direct lighting in rasterization-based graphics. In the Phong reflection model, the reflected light $\mathbf{R}$ is the sum of three modes: **ambient reflection**, **diffuse reflection** and **specular reflection**:

$$\mathbf{R} = \mathbf{R}_{\text{ambient}} + \mathbf{R}_{\text{diffuse}} + \mathbf{R}_{\text{specular}} \tag{10.3}$$

10.3.1 Ambient reflection

The ambient reflection is an approximation to the effect of global illumination. After the light reflects in a scene of many times, there is approximately an in-coming light to the fragment from all directions. In particular, $\mathbf{R}_{\text{ambient}}$ is proportional to the original light intensity $\mathbf{L}$, independent of the distance d, and is independent of $\mathbf{l}, \mathbf{n}, \mathbf{v}$. The ambient reflection formula is given by

$$\mathbf{R} = \mathbf{C}_{\text{ambient}}\mathbf{L} \tag{10.4}$$

where $\mathbf{C}_{\text{ambient}}$ is a material color.

10.3.2 Diffuse reflection

Diffuse reflection is a model for light reflections on a rough surface. The light comes onto the surface, and is "re-emit" back to the space in *all directions uniformly*. In particular, the reflected color is independent of the viewing direction $\mathbf{v}$.

The diffuse reflection is nevertheless a function of $\mathbf{l}$ and $\mathbf{n}$. This is because the re-emitted light is proportional to the amount of in-coming light per unit area. If the in-coming light $\mathbf{L}$ is the amount of light per unit area of the perpendicular cross-section of the light beam, then the amount of in-coming light per unit area on the surface is scaled by $\mathbf{l} \cdot \mathbf{n}$. This is called **Lambert's cosine law**.

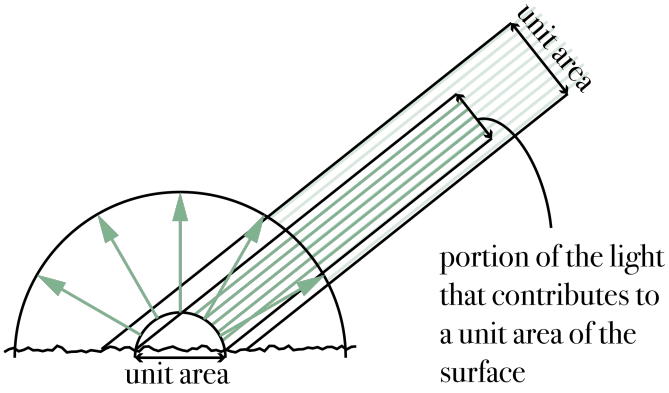


Figure 10.3 Incidence between a rough surface and a light beam.

Let us derive the diffuse reflection step-by-step. Suppose $\mathbf{L}$ is the intensity original light source. Let d be the distance between the light source and the surface. Let δ be the distance between the viewer and the surface. When the light arrives at the surface, the amount of light per unit cross-section area is given by

$$\begin{array}{l}\text{in-coming light per}\\ \text{cross-section area}\\ \text{at the surface}\end{array} = \frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.5}$$

where c_0, c_1, c_2 are some constants. For example, $\frac{\mathbf{L}}{d^2}$ describes that the light rays in the beam radiates from a point, so that the light per unit cross-section decays by an inverse-square law in distance. For another example, $c_0 = 1, c_1 = c_2 = 0$ describes that the light rays are parallel in the beam.

Now, per unit area on the surface, the amount of light is scaled by a cosine factor

$$\begin{array}{l}\text{in-coming light per}\\ \text{surface area}\end{array} = (\mathbf{n} \cdot \mathbf{l}) \frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.6}$$

Next, the surface reflects the light with a color mask $\mathbf{C}_{\text{diffuse}}$ as part of the property of the material color. The reflected light is re-emitted in all directions uniformly; that is, it is a radially spread-out light

$$\begin{array}{l}\text{amount of radial light}\\ \text{(re)emitted per surface area}\end{array} = \mathbf{C}_{\text{diffuse}} (\mathbf{n} \cdot \mathbf{l}) \frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2}. \tag{10.7}$$

Similarly, the out-going light that connects to the viewer has a light intensity per cross-section scaled by a cosine factor:

$$\begin{array}{l}\text{out-going light towards } \mathbf{v}\\ \text{per cross-section area}\\ \text{at the surface}\end{array} = (\mathbf{n} \cdot \mathbf{v}) \mathbf{C}_{\text{diffuse}} (\mathbf{n} \cdot \mathbf{l}) \frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.8}$$

Note that this reflected light is a radially fanned out light. As it reaches the camera, the light has attenuated by the inverse-squared law.

$$\begin{matrix}\text{out-going light towards } \mathbf{v} \\ \text{per cross-section area} \\ \text{at the camera}\end{matrix} = \frac{1}{\delta^2}(\mathbf{n}\cdot\mathbf{v})\mathbf{C}_{\text{diffuse}}(\mathbf{n}\cdot\mathbf{l})\frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.9}$$

For a fragment corresponding to a pixel, the physical area of the surface covered by the pixel is

$$\text{physical area of fragment} = \frac{(\text{pixel area})\,\delta^2}{\mathbf{n}\cdot\mathbf{v}}. \tag{10.10}$$

Hence

$$\mathbf{R}_{\text{diffuse}} = \begin{matrix}\text{The total light received} \\ \text{per pixel area}\end{matrix} = \frac{\delta^2}{\mathbf{n}\cdot\mathbf{v}}\frac{1}{\delta^2}(\mathbf{n}\cdot\mathbf{v})\mathbf{C}_{\text{diffuse}}(\mathbf{n}\cdot\mathbf{l})\frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.11}$$

$$= \mathbf{C}_{\text{diffuse}}(\mathbf{n}\cdot\mathbf{l})\frac{\mathbf{L}}{c_0 + c_1 d^1 + c_2 d^2} \tag{10.12}$$

In particular, δ and $\mathbf{n}\cdot\mathbf{v}$ terms all cancel out.

One last thing: the reflection is activated only when $\mathbf{n}\cdot\mathbf{l} > 0$. When $\mathbf{n}\cdot\mathbf{l} < 0$, we have that the light is behind the surface and hence the part of the surface is in the shadow rather be in the light. So, we replace the $(\mathbf{n}\cdot\mathbf{l})$ factor by

$$\max(\mathbf{n}\cdot\mathbf{l}, 0) = \begin{cases}\mathbf{n}\cdot\mathbf{l}, & \mathbf{n}\cdot\mathbf{l} > 0 \\ 0, & \text{otherwise.}\end{cases} \tag{10.13}$$

Summary 10.2 The diffuse reflection from a single light source is given by

$$\mathbf{R}_{\text{diffuse}} = \mathbf{C}_{\text{diffuse}} \max(\mathbf{n}\cdot\mathbf{l}, 0) f(d)\mathbf{L}, \quad f(d) = \min\left(\frac{1}{c_0 + c_1 d + c_2 d^2}, 1\right) \tag{10.14}$$

where $\mathbf{C}_{\text{diffuse}}$ is a material color, $\mathbf{n}$ is the surface normal, $\mathbf{l}$ is the unit vector pointing from the surface to the light source, $\mathbf{L}$ is the color of the light source, d is the distance between the surface and the light source, and (c_0, c_1, c_2) are parameters characterizing the attenuation profile as the light travels from the light source to the surface.

In practice, the attenuation factor $f(d)$ is wrapped by $\min(\cdot, 1)$ to prevent it from being greater than 1.

10.3.3 Blinn–Phong specular reflection

Specular reflection is like the mirror reflection, where the reflection is seen only when $\mathbf{v}, \mathbf{n}, \mathbf{l}$ lie in the same plane, and $\mathbf{v}$ makes the same angle with $\mathbf{n}$ as $\mathbf{l}$ makes an angle with $\mathbf{n}$. That is, $\mathbf{v}$ is the reflection of $\mathbf{l}$ about $\mathbf{n}$. For most of the non-perfect reflector, the specular reflection is not a perfect mirror. The specular reflection is a function of

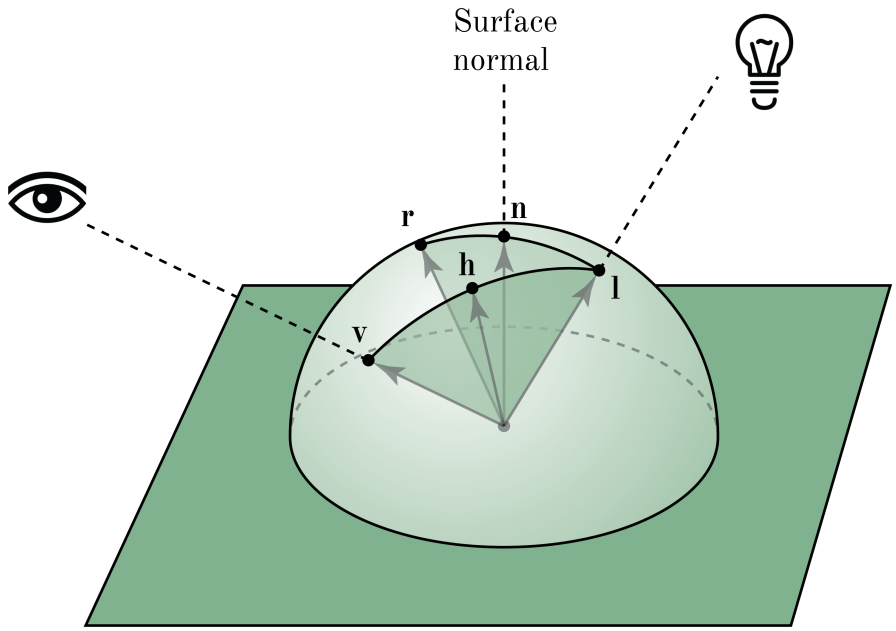


Figure 10.4 Unit vectors for reflection models.

$\mathbf{v}, \mathbf{n}, \mathbf{l}$ and we can already see a reflection when $\mathbf{v}$ is *close* to the reflected vector of $\mathbf{l}$ about $\mathbf{n}$.

A way to measure the closeness is to compute the true reflected vector of $\mathbf{l}$:

$$\mathbf{r} \coloneqq 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}, \tag{10.15}$$

and then measure the closeness in terms of the dot product $\mathbf{v} \cdot \mathbf{r}$. This is called **Phong specular reflection model**.

A slightly more efficient approach is **Blinn–Phong specular reflection model**. The closeness of $\mathbf{v}$ and the reflection direction $\mathbf{r}$ is replaced by

$$\mathbf{n} \cdot \mathbf{h} \tag{10.16}$$

where

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{|\mathbf{v} + \mathbf{l}|} \tag{10.17}$$

is called the half-way vector. It is the unit vector in the direction of the angle bisector between $\mathbf{v}$ and $\mathbf{l}$.

The efficiency of the Blinn–Phong specular model comes from the fact that when the camera and the lights are far away from the geometry, $\mathbf{v}, \mathbf{l}, \mathbf{h}$ are relatively constant across the geometry, and can sometimes be replaced by a constant, which is computed only once. On the other hand, an evaluation of $\mathbf{r}$ will have to be non-constant because it depends heavily on $\mathbf{n}$.

Now, the amount of reflection is a function of this closeness measurement $\max(\mathbf{n} \cdot \mathbf{h}, 0) \in [0, 1]$ (again we wrap it around with $\max(\cdot, 0)$ for the same reason as that in

diffuse reflection), which should be close to 1 when $\max(\mathbf{n}\cdot\mathbf{h},0)$ is close to 1, but drops down to near zero whenever $\max(\mathbf{n}\cdot\mathbf{h},0)$ is a bit off from 1. A simple function that fulfill this property is raising to a power $\sigma > 1$:

$$\max(\mathbf{n}\cdot\mathbf{h},0)^{\sigma}. \tag{10.18}$$

The larger σ is, the closer it gets to a perfect mirror reflection. The parameter σ is called the **shininess**.

Summary 10.3 The specular reflection from a single light source is approximately given by the Blinn–Phong model

$$\mathbf{R}_{\text{specular}} = \mathbf{C}_{\text{specular}} \max(\mathbf{n}\cdot\mathbf{h},0)^{\sigma} f(d)\mathbf{L}, \quad f(d) = \min\left(\frac{1}{c_0 + c_1 d + c_2 d^2}, 1\right). \tag{10.19}$$

Here, $\mathbf{C}_{\text{specular}}$ is specular reflection color (typically $\mathbf{C}_{\text{specular}} = (1,1,1)$ for insulator and some other colors for metals), $\mathbf{n}$ is the surface normal, $\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{|\mathbf{v}+\mathbf{l}|}$ is the unit vector angle-bisecting the unit vector $\mathbf{l}$ pointing from the surface to the light source and the unit vector $\mathbf{v}$ pointing from the surface to the viewer. Similar to diffuse reflection, $\mathbf{L}$ is the color of the light source, d is the distance between the surface and the light source, and (c_0, c_1, c_2) are parameters characterizing the attenuation profile as the light travels from the light source to the surface.

10.3.4 Multiple light sources

When there are multiple light sources, the reflection light is given by the sum of the reflections contributed from each light.

Summary 10.4 Let $i = 1, 2, \ldots, m$ be the index for each light source. Then the reflection formula is given by

$$\mathbf{R} = \underbrace{\mathbf{E}}_{\text{emission}} + \sum_{i=1}^{m} \Big[\mathbf{R}_{\text{ambient}}\mathbf{L}_i + \Big(\mathbf{R}_{\text{diffuse}} \max(\mathbf{n}\cdot\mathbf{l}_i,0) + \mathbf{R}_{\text{specular}} \max(\mathbf{n}\cdot\mathbf{h}_i)^{\sigma}\Big) f_i(d_i)\mathbf{L}_i\Big] \tag{10.20}$$

Emission can be nonzero if the surface glows itself.

10.4 Light at infinity

A light can be placed in the plane at infinity. In that case, the light represents a distant light shedding parallel light rays. A light position is represented in homogeneous coordinates

$$\mathbf{p}^{\text{light}} = \begin{bmatrix} p_x^{\text{light}} \\ p_y^{\text{light}} \\ p_z^{\text{light}} \\ p_w^{\text{light}} \end{bmatrix} \tag{10.21}$$

The light is at a finite position if $p_w^{\text{light}} \neq 0$, and the 3D position is given by $(p_x/p_w, \ldots, p_z/p_w)^{\text{light}}$. The light is at infinity if $p_w^{\text{light}} = 0$.

The only time one needs to use the light position is to evaluate the light direction vector $\mathbf{l} \in \mathbb{R}^3$ from a surface position $\mathbf{p}^{\text{surface}} \in \mathbb{R}^4$ to the light position $\mathbf{p}^{\text{light}}$. A naive approach is to evaluate

$$\mathbf{l} = \text{normalize}\left(\frac{1}{p_w^{\text{light}}} \begin{bmatrix} p_x^{\text{light}} \\ p_y^{\text{light}} \\ p_z^{\text{light}} \end{bmatrix} - \frac{1}{p_w^{\text{surface}}} \begin{bmatrix} p_x^{\text{surface}} \\ p_y^{\text{surface}} \\ p_z^{\text{surface}} \end{bmatrix} \right). \tag{10.22}$$

However, it is likely to give a division by zero. Technically, it is possible to set an "if" condition that asks if p_w^{light} is too small. But an artificial threshold is not very elegant. A more elegant evaluation is to combine the two fractions over a common denominator $p_w^{\text{light}} p_w^{\text{surface}}$; the common denominator can be removed in the normalization function

$$\mathbf{l} = \text{normalize}\left(p_w^{\text{surface}} \begin{bmatrix} p_x^{\text{light}} \\ p_y^{\text{light}} \\ p_z^{\text{light}} \end{bmatrix} - p_w^{\text{light}} \begin{bmatrix} p_x^{\text{surface}} \\ p_y^{\text{surface}} \\ p_z^{\text{surface}} \end{bmatrix} \right). \tag{10.23}$$

The formula is stable around $p_w^{\text{light}} \approx 0$.

11. Interpolations

Without interpolation all science would be impossible.

Henri Poincaré, 1907

Linear interpolations play an important role in computer graphics. Linear interpolations are also called **lerps** in some communities. In ancient history of mathematics, linear interpolations are used to evaluate the value of a function at an arbitrary input while the function values are only known at a few points in a look-up table, *e.g.* trigonometry table. In graphics, we often encounter a similar task. For example, the value of vertex attributes are known only at triangle vertices, yet the fragment is somewhere in between those vertices. In the next chapter about textures, we will also need to look up a texture color from an image, but the texture coordinate is not necessarily right on an image pixel of the image.

In this chapter, we will look at the linear interpolation and some variants of it.

11.1 Barycentric coordinates

Barycentric coordinates were introduced by August Möbius in 1827 together with his introduction of homogeneous coordinates. Therefore, in the following, we will see some ideas reminiscent to homogeneous coordinates used in affine geometry.

The discussion about barycentric coordinates and barycenters will be taken under the context of affine geometry. See Section 7.2 for the relevant definitions in affine geometry.

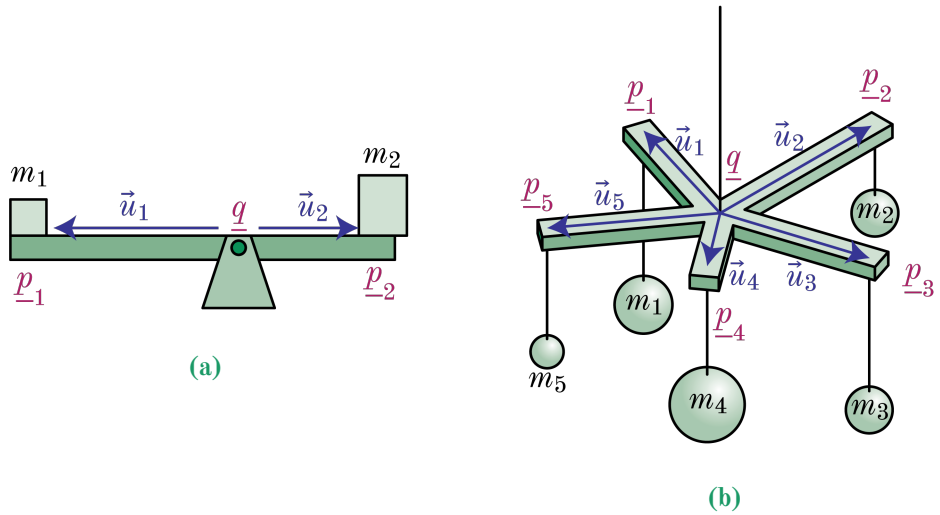


Figure 11.1 There is a unique position $\underline{q}$ of the fulcrum for a lever to be in balance.

11.1.1 Archimedes' law of lever

Consider a one-dimensional lever pivoting on the fulcrum at $\underline{q}$ with two masses m_1, m_2 located at $\underline{p}_1, \underline{p}_2$ respectively, as shown in Figure 11.1 (a). The displacement from the fulcrum to the masses are denoted by $\vec{u}_i = \underline{p}_i - \underline{q}$. In Figure 11.1 (a), the two displacements are pointing in directions negative of each other. A **moment** is a displacement scaled by the mass. Here, we have two moments:

$$m_1\vec{u}_1, \quad m_2\vec{u}_2. \tag{11.1}$$

Archimedes' law of lever states that *the lever is in equilibrium when the sum of moments is zero*:

$$\text{Figure 11.1 (a) is in equilibrium} \quad \Leftrightarrow \quad m_1\vec{u}_1 + m_2\vec{u}_2 = \vec{0}. \tag{11.2}$$

In fact, Archimedes' law of lever applies to a balance loaded with an arbitrary number of masses in dimensions greater or equal to one. Let P be an affine space (of positions) associated with a real vector space V. Let $m_1, \ldots, m_N \in \mathbb{R}$ be masses located at $\underline{p}_1, \ldots, \underline{p}_N \in P$. Let $\vec{u}_i = \underline{p}_i - \underline{q} \in V$ denote their displacement vectors from a fulcrum at $\underline{q} \in P$. For example, Figure 11.1 (b) shows a two-dimensional P with 5 masses. The system is in equilibrium if the sum of moments vanishes

$$\sum_{i=1}^{N} m_i\vec{u}_i = \vec{0}. \tag{11.3}$$

Theorem 11.1 Consider masses $m_1, \ldots, m_N \in \mathbb{R}$ located at $\underline{p}_1, \ldots, \underline{p}_N \in P$ respectively. If the total mass $m_1 + \cdots + m_N$ is nonzero, then there is a unique position $\underline{q}$ to place the fulcrum so that the lever system is in equilibrium. The solution is

given by

$$\underline{q} = \frac{m_1 \underline{p}_1 + \cdots + m_N \underline{p}_N}{m_1 + \cdots + m_N}. \tag{11.4}$$

Proof. Substituting $\vec{u}_i = \underline{p}_i - \underline{q}$ into the condition $\sum_{i=1}^N m_i \vec{u}_i = \vec{0}$ yields

$$\vec{0} = \sum_{i=1}^N m_i (\underline{p}_i - \underline{q}) \tag{11.5}$$

$$= \sum_{i=1}^N m_i \underline{p}_i - \left(\sum_{i=1}^N m_i \right) \underline{q}, \tag{11.6}$$

which is equivalent to (11.4). □

11.1.2 Barycenter

Definition 11.1 — Barycenter. The unique location for the fulcrum (11.4) is called the **barycenter** of the points $\underline{p}_1, \ldots, \underline{p}_N$ with weights $m_1, \ldots, m_N$.

If two sets of weights have the same proportion

$$\mathbf{m} = \begin{bmatrix} m_1 \\ \vdots \\ m_N \end{bmatrix} \sim \begin{bmatrix} \alpha m_1 \\ \vdots \\ \alpha m_N \end{bmatrix} = \alpha \mathbf{m} \tag{11.7}$$

then they give rise to the same barycenter (the constant scaling factor α cancels out in (11.4)).

11.1.3 Affine combinations

In (11.4), the affine points $\underline{p}_1, \ldots, \underline{p}_N$ are taken a linear combination. This linear combination is generally not allowed since addition between two affine points are meaningless. A linear combination as such makes sense only when the coefficients $\frac{m_i}{m_1 + \cdots + m_N}$ sum up to 1.

Definition 11.2 — Affine combination. A linear combination of vectors or affine points

$$\lambda_1 \underline{p}_1 + \cdots + \lambda_N \underline{p}_N \tag{11.8}$$

is called an **affine combination** if

$$\lambda_1 + \cdots + \lambda_N = 1. \tag{11.9}$$

Affine combinations are weighted averages of points.

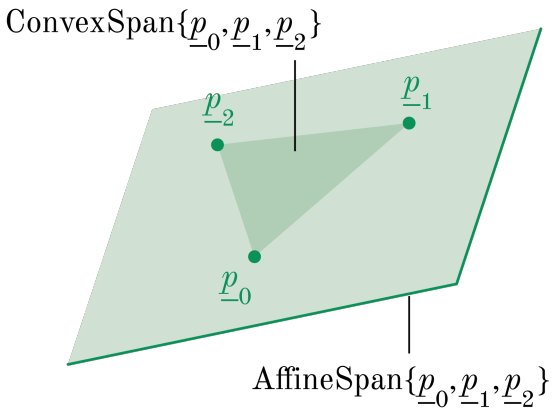


Figure 11.2 The affine span of 3 points is a 2D plane. The convex span of 3 points is the triangle formed by the 3 points lying in the 2D plane.

Summary 11.1 An affine combination of points is the barycenter of the point assigned with a list of masses in the same proportion as the affine combination coefficients.

A special type of affine combination is when all the weights are non-negative.

Definition 11.3 — Convex combination. An affine combination of points

$$\lambda_1 \underline{p}_1 + \cdots + \lambda_N \underline{p}_N, \quad \lambda_1 + \cdots + \lambda_N = 1 \tag{11.10}$$

is called a **convex combination** if $\lambda_i \geq 0$ for all $i = 1, \ldots, N$.

Definition 11.4 — Affine independence. Let P be an affine space associated with a vector space V. $(k+1)$ points $\underline{p}_0, \ldots, \underline{p}_k \in P$ are said to be **affinely independent** if the k displacement vectors $(\underline{p}_1 - \underline{p}_0)$, …, $(\underline{p}_k - \underline{p}_0) \in V$ from one of the points $(\underline{p}_0)$ are linearly independent.

If $\underline{p}_0, \ldots, \underline{p}_k \in P$ are affinely independent, we also say that $\underline{p}_0, \ldots, \underline{p}_k$ are the vertices of a non-degenerate ***k*-simplex**.

▪ **Example 11.1** Two non-coinciding points are affinely independent. They are the vertices of a line segment (1-simplex). ▪

▪ **Example 11.2** Three points are affinely independent if and only if they are not collinear. They form a non-degenerate triangle (2-simplex). ▪

▪ **Example 11.3** Four points in 3D that are not coplanar are affinely independent. They form a non-degenerate tetrahedron (3-simplex). ▪

▪ **Example 11.4** The maximal number of points in an n-dimensional affine space that can be affinely independent is $(n+1)$. The $(n+1)$ points constitute the vertices of an n-simplex. ▪

The affine combinations of $(k+1)$ affinely independent points span the k-dimensional

plane that pass through the simplex.

▪ **Example 11.5** 3 affinely independent points (forming a triangle) affinely span the 2-dimensional plane in which the triangle lies. ▪

▪ **Example 11.6** 2 affinely independent points (forming a line segment) affinely span the straight line containing the two points. ▪

The *convex* combination of $(k+1)$ affinely independent points span the interior of the k-simplex.

> **Summary 11.2** $(k+1)$ affinely independent points form a non-degenerate k-simplex in an n-dimensional affine space, $k \leq n$. Affine combinations of these $(k+1)$ points span the k-dimensional plane that contains all the $(k+1)$ points. Convex combinations of the $(k+1)$ points span the simplex.

11.1.4 Barycentric coordinates

Suppose $\underline{p}_0, \ldots, \underline{p}_k$ are affinely independent. Then for each point $\underline{q}$ in the affine span of $\underline{p}_0, \ldots, \underline{p}_k$, there is a *unique* set of coefficients

$$\boldsymbol{\lambda} = \begin{bmatrix} \lambda_0 \\ \vdots \\ \lambda_k \end{bmatrix} \in \mathbb{R}^{k+1}, \quad \lambda_0 + \cdots + \lambda_k = 1, \tag{11.11}$$

such that

$$\underline{q} = \begin{bmatrix} \underline{p}_0 & \cdots & \underline{p}_k \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \vdots \\ \lambda_k \end{bmatrix}. \tag{11.12}$$

The coefficients $\boldsymbol{\lambda} \in \mathbb{R}^{k+1}$ are called the **barycentric coordinates** of $\underline{q}$ with respect to the points $\underline{p}_0, \ldots, \underline{p}_k$.

11.2 Linear interpolation in a simplex

Suppose we have $(k+1)$ affinely independent points $\underline{p}_0, \ldots, \underline{p}_k$. On top of these points, we assign "vertex attributes" $f_0, \ldots, f_k$. The values of $f_0, \ldots, f_k$ can be in $\mathbb{R}$, in $\mathbb{R}^m$, or generally in some other affine space. For example, f's can be color, and the space of RGB colors can be viewed as an affine space, in which blending colors by affine combinations is a meaningful action.

Now, with the given position-value data $(\underline{p}_0, f_0), \ldots, (\underline{p}_k, f_k)$, there is a unique affine function f defined over the kD plane affinely spanned by $\underline{p}_0, \ldots, \underline{p}_k$ so that $f(\underline{p}_i) = f_i$ for all $i = 0, \ldots, k$. This interpolated affine function f is given by the following. For each $\underline{q}$ in the affine span of $\underline{p}_0, \ldots, \underline{p}_k$, let $(\lambda_0, \ldots, \lambda_k)$ be the barycentric coordinates. Then we let

$$f(\underline{q}) \coloneqq \lambda_0 f_0 + \cdots + \lambda_k f_k. \tag{11.13}$$

We call (11.13) an **affine interpolation**, or more often called a **linear interpolation**, among the vertices of a simplex.

11.2.1 Linear interpolation between two points

Let $x_0, x_1 \in \mathbb{R}$ be two positions on the real line. Suppose f_0 is a data value assigned on x_0 and f_1 is a data assigned on x_1. Now, given an arbitrary $x \in \mathbb{R}$, we should be able to evaluate $f(x)$ with linear interpolation.

First, the barycentric coordinates (λ_0, λ_1) of x with respect to x_0, x_1 are computed by the following conditions

$$\begin{cases} \lambda_0 x_0 + \lambda_1 x_1 = x \\ \lambda_0 + \lambda_1 = 1. \end{cases} \tag{11.14}$$

The solution is given by

$$\lambda_0 = \frac{x_1 - x}{x_1 - x_0}, \quad \lambda_1 = \frac{x - x_0}{x_1 - x_0}. \tag{11.15}$$

So,

$$f(x) = \frac{x_1 - x}{x_1 - x_0} f_0 + \frac{x - x_0}{x_1 - x_0} f_1. \tag{11.16}$$

11.2.2 Linear interpolation of a triangle in a plane

A common task is the following.

Task 11.2 Suppose we have a triangle in the 2D screen with vertices (a_x, a_y), (b_x, b_y), (c_x, c_y). Now suppose we have a point (q_x, q_y) in the screen. Find the barycentric coordinate for (q_x, q_y) with respect to the triangle vertices.

The scenario takes place during rasterization. Given a fragment at pixel (q_x, q_y) from triangle (a_x, a_y), (b_x, b_y), (c_x, c_y), the rasterizer internally computes the barycentric coordinates.

To solve Task 11.2, let $\lambda_a, \lambda_b, \lambda_c \in \mathbb{R}$ be the unknown variables. They must satisfy

$$\lambda_a \begin{bmatrix} a_x \\ a_y \end{bmatrix} + \lambda_b \begin{bmatrix} b_x \\ b_y \end{bmatrix} + \lambda_c \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} q_x \\ q_y \end{bmatrix}, \quad \text{and} \quad \lambda_a + \lambda_b + \lambda_c = 1. \tag{11.17}$$

This corresponds to the system of linear equations for λ's:

$$\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_a \\ \lambda_b \\ \lambda_c \end{bmatrix} = \begin{bmatrix} q_x \\ q_y \\ 1 \end{bmatrix}, \tag{11.18}$$

which can be solved with a matrix inversion.

During rasterization, the vertex attributes f_a, f_b, f_c are linearly interpolated to an arbitrary point $\underline{q}$. They are interpolated precisely by the affine combination using this barycentric coordinate $(\lambda_a, \lambda_b, \lambda_c)$:

$$f(\underline{q}) = \lambda_a f_a + \lambda_b f_b + \lambda_c f_c. \tag{11.19}$$

Note 11.1 *The rasterization performs this linear interpolation in the screenspace. If the projection from the world to the screen is an orthographic (affine) projection, then this interpolation is consistent with performing the affine interpolation first in the world and then projecting the result into the pixel. However, if the projection from the world to the screen is perspective, then the this screen-space linear interpolation in the rasterization process will be different from the result of first affine interpolating the data in the world and then project to the pixel.*

Additional mathematical tricks need to be employed to let the the built-in interpolation in the rasterizer to be not only affine but also projective.

IV Appendix

A. Compilation and Linking

After innumerable failures I finally uncovered the principle for which I was searching, and I was astounded at its simplicity.

Alexander Graham Bell

Here, we give a quick review on building a C++ program. We assume that the readers have some C++ programming experience. Perhaps, in their first course in learning C++, the programs were small and self-contained, or that the project was handled by a project managing software like Visual Studio, or that a makefile was given by the instructor; the C++ programming experience did not imply a working knowledge of building a larger project and linking libraries from scratch. Hence, the purpose of this appendix is to help those readers feel comfortable with what to expect in a complex C++ project.

A.1 From source code to executable

The process of turning a **source code** (the C++ code we write) into an **executable** (**machine code**, *i.e.* **binaries**) is called a **build**. A build consists of two stages: **compilation** and **linking**.

Compilation is to translate the source code into parts of the final machine code. These pieces of machine code produced by the compilation stage are called **object files** (`.o` files). Linking, on the other hand, is to combine one or multiple object files into the final executable.

For example, suppose we have a source code `A.cpp`. Then

```
# compilation
    >> g++ -c A.cpp
```

produces a new file "A.o." Note that "-c" indicates that this is a pure compilation without linking. Next,

```
# linking
    >> g++ -o MyExecutable A.o
```

will link "A.o" and output an executable named "MyExecutable."

A.1.1 Multiple source code files

Compiling one .cpp file generates one .o file with the same name. In a typical project, there are many .cpp files, and thus the compilation produces multiple .o files. How does the linker know how to combine these .o files?

Suppose in A.cpp, we want to call a function "void b(void)" defined in B.cpp. All we need to do is to declare "void b(void);" in the beginning of A.cpp. Then, the compiler will produce an A.o which knows that the machine code corresponding to the subroutine b() is to be defined. It is at the linking stage that the subroutine b() is filled from B.o.

To make life easy, instead of carefully declaring "void b(void)" in A.cpp, we write a **header file** B.h that contains the declaration "void b(void);." The content of the header file B.h is to be copied (by calling "#include "B.h"") to other files such as A.cpp.

The header file is the interface for other files to include.

To build the program:

```
#compilation stage
    >> g++ -c A.cpp
    >> g++ -c B.cpp
#linking stage
    >> g++ -o MyExecutable  A.o B.o
```

In case B.h is hidden in some other directory, we can either hardcode the path in the source code (writing "#include "./path/to/directory/containing_B_h/B.h"" in A.cpp), or keep the line "#include "B.h"" clean, and build the program by adding the search directory (using the "-I" flag):

```
#compilation stage
    >> g++ -I./path/to/directory/containing_B_h/ -c A.cpp
    >> g++ -c B.cpp
#linking stage
    >> g++ -o MyExecutable  A.o B.o
```

Summary A.1 A C++ compiler such as g++, clang, Visual C++, *etc.*, is used as both a compiler and a linker. The option -c tells the compiler to perform only the compilation stage, which translates each .cpp code into a .o machine code called an object file. The option -I is a *compilation option* that includes the search path for the headers. The option -o is a *linker option* for the output filename.

Headers containing function definitions

Sometimes, a header file `B.h` contains not only a list of function declarations, but also the function definition itself. This scenario usually happens when the function definition is as short as a one-liner, and the programmer is too lazy to write the function definition in the `B.cpp` file. This is fine, until `B.h` is included multiple times, causing the error of some functions defined multiple times. Therefore, for header files containing function definitions (instead of declaration-only), the code is wrapped by

Code A.1 `B.h`

```
#ifndef __B_HEADER
#define __B_HEADER

    // The actual header code

#endif
```

to avoid repeated inclusion of the header code.

Tip If your header contains function definitions, and if the header may potentially be included multiple times, wrap the header code like Code A.1. See also **One Definition Rule**.[1]

A.2 Library

A final program may contain many subroutines (in machine binaries) that are common across different programs and projects. Thus, it is a good idea to share these subroutines across different builds. A collection of these subroutines with well-understood functionalities and well-defined interface is called a **library**. The linking stage can also gather library files (in addition to just the object files). The linking stage can also produce library files (instead of an executable).

Earlier, we see the example with "`B.h`" as the interface, and "`B.o`" as the object file to be linked with "`A.o`." Libraries work similarly. The interface of a library is a (or multiple) header file(s). It contains a list of function declarations. Suppose we want to use functions in the `XXX` library:

- Include `#include <path/to/the/library/XXX.h>`[2] in our `A.cpp` file. Then, the compilation produces an `A.o` which knows that the machine code corresponding to the subroutines from the library are to be filled.
- At the linking stage, in addition to linking our own object files (such as `A.o`, `B.o`), link the library files that contains the machine code.

Summary A.2 A library consists of (i) header files to be included; and (ii) binaries, understood as shared object files, to be linked.

[1] `https://en.wikipedia.org/wiki/One_Definition_Rule`

[2] For more information of `#include <...>` v.s. `#include "..."`, see `https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html`

	static library	dynamic library
Windows	`XXX.lib`	`XXX.dll` (dynamic-link library)
Linux	`libXXX.a`	`libXXX.so` (shared object)
MacOS	`libXXX.a`	`libXXX.dylib` (dynamic-link library), or other filenames containing `XXX` placed in a macOS's "`.framework/`" structured folder.

Table A.1 File extensions of library binaries that are to be linked statically or dynamically.

The inclusion of the library headers is the same as including ordinary headers:

```
# compilation
    >> g++ -I path/to/the/library/header/ -c A.cpp
```

Then we link the library just like how we link object files earlier:

Code A.2

```
# linking
    >> g++ -o MyExecutable A.o path/to/the/library/binaries/libXXX.a
```

Now, the actual library binary file has filenames depending on the operating system and the mode (static or dynamic) of the linking (Table A.1). (We will explain static/dynamic linking later.) To suppress the file extension, we use the "`-l`" option for linking a library. Code A.2 is equivalent to

```
# linking
    >> g++ -o MyExecutable A.o -lpath/to/the/library/binaries/XXX
```

A convenient flag is "`-L`" which looks in directory for library files. Code A.2 is equivalent to

```
# linking
    >> g++ -o MyExecutable A.o -Lpath/to/the/library/binaries/ -lXXX
```

Summary A.3 "`-l`" is a linker option that links with a library file. "`-L`" is a linker options for looking in directory for library files. The prefix "`lib`" and the file extension in the filename "`libXXX.a`" do not need to be included when using `-l`.

Tip Put the "`-L`[directory]" before "`-l`[library]," otherwise the linker wouldn't know where to find the library if the search path has not been added.

A.2.1 Static and dynamic library linking

There are two different ways we can fill the subroutine binaries from the library: **static** or **dynamic**. Whether to link the library statically or dynamically is inferred from the file extension of the library file name (Table A.1).

> **Definition A.1** A **static linking** copies the library binaries into our executable. A **dynamic linking** has the library binaries called only at run time.

Consequently, the executable using statically linked libraries is self-contained (static build). This executable can run on other computer of the same operating system without additional dependency, since the relevant binaries have already been embedded in the executable.

On the other hand, an executable involving dynamic link will require the library files with the correct version and in the correct path on the machine.

A library can easily be made into a "standalone" library. Simply put those `.lib`, `.dll` (Windows) or `.a`, `.so`, `.dylib` (Linux/MacOS) files in the local directory of the project, conventionally in a folder named `lib/`. The headers for the library are conventionally placed in a folder named `include/`.

Tip When a directory `path/to/lib/bin` contains both a static and a dynamic library of the same name, say "`libXXX.a`" and "`libXXX.so`." Then

```
-Lpath/to/lib/bin -lXXX
```

will take the dynamic one ("`.so`") by default.

Tip It could be dangerous to accept a standalone, pre-built library with only the headers and the binaries. The machine code without a transparent source code from an unknown developer or distributor can do harmful thing to the computer without letting you know.

A.2.2 Mac's framework directory

On a Mac computer, sometimes a dynamic library is located in a structured `.framework` directory, such as in `path/to/frameworks/XXX.framework/`. This `XXX.framework` directory contains both the library *binaries* and the *headers*. If the library is packaged in such a framework, we include the header search path using "`-F`" (instead of using "`-I`") and link the library using "`-framework`" (instead of using "`-L`" and "`-l`")

```
# compilation
    >> g++ -F path/to/frameworks/XXX -c A.cpp
# linking
    >> g++ -o MyExecutable A.o -framework path/to/frameworks/XXX
```

If the "`XXX`" is a commonly used library already installed by Apple's Software Development Kit (SDK), then the search path of its header is already added by the system. In that case, the following would work:

```
# compilation
    >> g++ -c A.cpp
# linking
    >> g++ -o MyExecutable A.o -framework XXX
```

A.3 Makefile and Integrated Development Environment (IDE)

Now that we have learned the basic principles of compilation and linking, we can start organizing a project. A project potentially consists of many source code and headers, using various libraries. In order to organize them, we write a `makefile` or we use an integrated development environment (IDE) such as Visual Studio, Xcode, Eclipse, *etc*.

Task A.1 Suppose we want to build `myExecutable` from the source code `A.cpp`, using functions declared in `./include/B.h` and defined in `./src/B.cpp`. Suppose one or all of `A.cpp`, `B.h`, `B.cpp` are using a library `XXX` with its header located in the directory "`path/to/lib/include`" and with its binary located in the directory "`path/to/lib/bin`." Suppose our C++ requires C++11 support.

Construct a makefile or a project in an IDE that records the above specification.

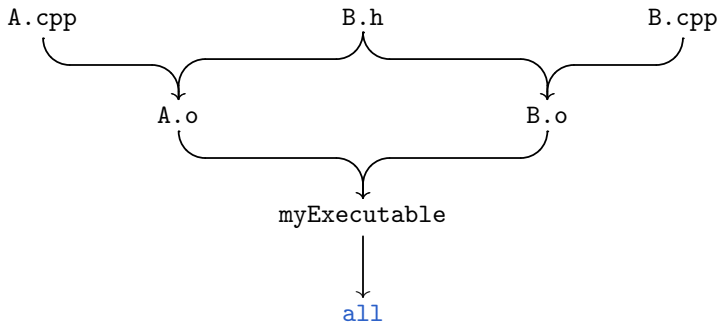
A.3.1 Makefile

A makefile is a plain text file named `makefile` that specifies command routines. Here, the command routines are compilation and linking.

Code A.3 `makefile`

```
# definitions for string replacement
CC = g++ # C++ compiler
CFLAGS = -std=c++11 # support c++11
INCFLAGS = -I./include -Ipath/to/lib/include
LDFLAGS = -Lpath/to/lib/bin -lXXX
RM = /bin/rm -f  # remove file command in linux and MacOS

all: myExecutable
myExecutable: A.o B.o
    $(CC) -o myExecutable A.o B.o $(LDFLAGS) #linking
A.o: A.cpp B.h
    $(CC) $(CFLAGS) $(INCFLAGS) -c A.cpp #compilation
B.o: B.cpp B.h
    $(CC) $(CFLAGS) $(INCFLAGS) -c B.cpp #compilation

clean:
    $(RM) *.o myExecutable
```

Here, the colon ":" in the makefile describes the dependency between "nodes" forming a network:



(A.1)

Tip Draw the dependency network of your project first, and then write the makefile according to it.

A.3.2 Using an IDE

Index