Computer Graphics

Karin Kosina (vka kyrah)
OpenGL
Open Graphics Library
OpenGL

• a platform-independent API for 2D and 3D graphics applications
• a standard, not a library
  • various implementations (e.g. by graphics card vendors) with varying degrees of optimisation
• Input: primitives (polygons, lines, points)
• Output: pixels
The Rendering Pipeline
Concepts

• Pipelines? Think of oil pipelines, assembly lines, ski lifts,...
• Pipelines consist of stages.
  • In an oil pipeline, the oil passes through sequentially.
  • The speed of the pipeline is determined by the slowest part of the pipeline, no matter how fast the other stages may be.
• Ideally, a pipeline of $n$ stages should give a speed-up of factor $n$
  • assembly line is a good example
Concepts

• Pipeline stages are executed in parallel, but they are stalled until the slowest stage has finished its task.

• cf. a car factory assembly line:
  • attaching the steering wheel takes 3 minutes
  • each other step takes 2 minutes
  • ➔ you can finish one car every 3 minutes

• Slowest stage = “bottleneck”
Graphics Rendering Pipeline

- Function:
  - generate ("render") a 2-dimensional image given 3-dimensional objects (and a virtual camera, light sources, a lighting model, etc.)

- Rendering speed
  - update speed of images
  - expressed in frames per second (fps)
  - rendering speed is determined by the bottleneck
Overview

Application → Geometry → Rasterizer
Overview

Application → Geometry → Rasterizer
The Application Stage

- Fully controlled by application programmer
  - collision detection,
  - input handling (keyboard, mouse, any other devices)
  - animations (updating model transformations)
  - acceleration algorithms (such as hierarchical view frustum culling)
- Output:
  - Geometry to be rendered in the form of rendering primitives (points, lines, triangles)
The Geometry Stage

- Computes what should be drawn, where it should be drawn, how it should be drawn.
- Handles per-vertex operations.
- Can be subdivided into five functional stages:
  - model & view transform, lighting, projection, clipping, screen mapping.
- With a single light source, each vertex requires approximately 100 individual floating point operations!
Overview

Application → Geometry → Rasterizer
The Rasterization Stage

- Input: transformed and projected vertices, colors, and texture coordinates from the geometry stage.
- Task is to assign correct colors to the pixels on the screen to render a correct image.
- **Rasterization (aka scan conversion):**
  - Conversion of 2d vertices in screen space (each with a z-value, one or two colors, and possibly a set of texture coordinates) into pixels on the screen.
The Rasterization Stage

- Handles per-pixel operations.
- Information for each pixel is stored in the color buffer (a rectangular array of colors).
- Color buffer should contain only the colors of the primitives which are visible from the point of view of the camera.
- This is usually done using the Z-Buffer algorithm.
Summary

Application ➔ Geometry ➔ Rasterizer
OpenGL

- a platform-independent API for 2D and 3D graphics applications
- a standard, not a library
- various implementations (e.g. by graphics card vendors) with varying degrees of optimisation
- Input: primitives (polygons, lines, points)
- Output: pixels
- low-level
- state-machine
- only does rendering
- need additional framework for OS integration, image loading,...
STL?
STL?
Simple Directmedia Layer
SDL

- SDL is a free cross-platform multi-media development API
- abstraction for OS-dependent tasks
  - create window and rendering context
  - handle keyboard, mouse, and joystick events
- audio
- thread abstraction
- ...
- see http://libsdl.org
anatomy of an SDL application

1. Initialise SDL (SDL_Init())
2. Create OpenGL rendering context (SDL_SetVideoMode())
3. Do your own OpenGL and app initialisation
4. Run main loop:
   - rendering
   - event processing
5. Cleanup
brace yourselves
int main(int argc, char ** argv)
{
    int width = 640, height = 480;

    // Initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        fprintf(stderr, "Unable to init SDL: %s\n", SDL_GetError());
        return -1;
    }

    if (!SDL_SetVideoMode(width, height, 32, SDL_OPENGL)) {
        fprintf(stderr, "Unable set video mode: %s\n", SDL_GetError());
        SDL_Quit();
        return -1;
    }

    SDL_WM_SetCaption("SDL/OpenGL intro", NULL);  // window title
    myinit(width, height);  // initialize OpenGL

    // ... continued on next page
// main application loop
bool done = false;
while (!done) {
    mydisplay();
    SDL_Event event;
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT) done = true;
        if (event.type == SDL_KEYDOWN) {
            switch(event.key.keysym.sym) {
                case SDLK_ESCAPE:
                    done = true;
                    break;
            }
        }
    }
}

SDL_Quit();
return 0;
</SDL>
now for some OpenGL fun!
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}

OpenGL initialisation

```c
void myInit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up
    glMatrixMode(GL_MODELVIEW);
}
```
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
               0.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
               0.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
The Z-Buffer

• The Z-buffer is the same size as the color buffer and stores the z-value from the camera to the closest primitive.

• When a primitive is rendered to a certain pixel, the z-value of the primitive at that pixel is computed and compared to the contents of the Z-buffer at the same pixel.

  • If the new z value is smaller than the z value in the Z-buffer, the primitive is closer to the camera ➔ the z value and the color of that pixel are updated.

  • If the new z value is greater, color and z are not changed.
OpenGL initialisation

void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}

OpenGL initialisation
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
               0.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
               0.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
Two projection methods:
- orthographic vs. perspective projection
- Orthographic projection:
  - View volume is a rectangular box.
  - Parallel lines remain parallel after the transform.
The preceding paragraph mentions inches and millimeters - do these really have anything to do with OpenGL? The answer is, in a word, no. The projection and other transformations are inherently unitless. If you want to think of the near and far clipping planes as located at 1.0 and 20.0 meters, inches, kilometers, or leagues, it's up to you. The only rule is that you have to use a consistent unit of measurement. Then the resulting image is drawn to scale.

Orthographic Projection

With an orthographic projection, the viewing volume is a rectangular parallelepiped, or more informally, a box (see Figure 3-15). Unlike perspective projection, the size of the viewing volume doesn't change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

Figure 3-15 : Orthographic Viewing Volume

The command 

\texttt{glOrtho(float left, float right, float bottom, float top, float near, float far);}

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.

With no other transformations, the direction of projection is parallel to the z-axis, and the viewpoint faces OpenGL Programming Guide (Addison-Wesley Publishing Company)
projection
projection

• Perspective projection:
  • The farther away an object lies from the camera, the smaller it appears after projection.
  • Parallel lines converge at the horizon.
  • View volume (called frustum) is a truncated pyramid with a rectangular base.
glFrustum(float left, float right,
    float bottom, float top,
    float near, float far);

Figure 3-13: Perspective Viewing Volume Specified by glFrustum()
projection

```c
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);
```

For perspective projection, the projection matrix is determined using the angle of view (fovy) and the aspect ratio of the width to height (x/y). The near and far clipping planes also define the extent of the projection volume. The `gluPerspective()` function creates a perspective-view frustum matrix that can be multiplied with the current matrix to apply the transformation. The field of view (fovy) is specified in the x-z plane and must be within the range [0.0, 180.0]. The aspect ratio (aspect) is the ratio of the width to the height of the frustum, and the near and far values (near, far) define the distances from the viewpoint to the near and far clipping planes, along the negative z-axis. These values should always be positive.

In contrast to `glFrustum()`, `gluPerspective()` allows for rotations or translations to change the default orientation of the viewing volume, with no transformations, the viewpoint remains at the origin and the line of sight points down the negative z-axis. When using `gluPerspective()`, it is important to select appropriate values for the field of view to avoid distorted images. For example, to draw to the entire screen which happens to be 11 inches high, a field of view of 90 degrees would require the viewer to be about 7.8 inches from the screen for the image to appear undistorted. If the viewer is farther from the screen than usual, the perspective might not look right. When the drawing area occupies less than the full screen, the viewer needs to be even closer to maintain the correct field of view.

To achieve a perfect field of view, one must consider the actual distance from the eye to the screen and the size of the window, calculating the angle the window subtends at that size and distance. It is likely that this angle will be smaller than expected. Another way to think about it is that a 94-degree field of view with a 35-millimeter camera requires a 20-millimeter lens. More details on how to solve these issues can be found in the "Troubleshooting Transformations" section.
projection
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-3, 3, -3, 3, 2, 10);
    gluLookAt(0.0, 0.0, 8.0, // eye
        0.0, 0.0, -1.0, // center
        0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
World-space view

Screen-space view

Command manipulation window

```
fovy  aspect  zNear  zFar

gluPerspective( 60.0  , 1.00  , 1.0  , 10.0  );

  gluLookAt( 0.00  , 0.00  , 2.00  ,  <- eye

     0.00  , 0.00  , 0.00  ,  <- center

     0.00  , 1.00  , 0.00  );  <- up

Click on the arguments and move the mouse to modify values.
```
OpenGL initialisation

```c
void myinit(int width, int height) {
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
               0.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
OpenGL initialisation

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
                0.0, 0.0, -1.0, // center
                0.0, 1.0, 0.0); // up

    glMatrixMode(GL_MODELVIEW);
}
```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
```
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}

void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
Screen-space view

Command manipulation window

```c
glBegin (GL_TRIANGLES);
setColor3f (0.00, 0.00, 1.00);
Vertex2f (50.0, 50.0);
setColor3f (0.00, 0.50, 1.00);
Vertex2f (100.0, 150.0);
setColor3f (0.50, 0.50, 1.00);
Vertex2f (175.0, 175.0);
setColor3f (0.50, 0.00, 1.00);
Vertex2f (200.0, 100.0);
EndInit;
```

Click on the arguments and move the mouse to modify values.
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}

Double-Buffering

- To avoid visible flickering during the rasterization process, double buffering is used:
  - Rendering is done off-screen in the back buffer.
  - When the rendered scene is complete, front and back buffer are swapped.
  - The swapping is done during the vertical monitor sync, so that it is not visible.
let's move the triangle
modify drawing code

void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f, -1.0f, 0.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
modified drawing code

```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 1.0f, 1.0f, 0.0f);
    glVertex3f( 2.0f, -1.0f, 0.0f);
    glVertex3f( 0.0f, -1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
```
this works
but can get kinda tedious
there’s a better way
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    
    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();
    
    SDL_GL_SwapBuffers();
}
```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glTranslatef(1.0f, 0.0f, 0.0f);

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
```
and one more possibility
move the camera

```c
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 0.0, 4.0, // eye
              0.0, 0.0, -1.0, // center
              0.0, 1.0, 0.0); // up
    glMatrixMode(GL_MODELVIEW);
}
move the camera

```cpp
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(-1.0, 0.0, 4.0, // eye
               -1.0, 0.0, -1.0, // center
               0.0, 1.0, 0.0); // up
    glMatrixMode(GL_MODELVIEW);
}
```
a few words on coordinate systems
coordinate systems

- On the way to the screen, a model is transformed into several different spaces or coordinate systems:
  - model space
  - world space [result of model transform]
  - camera space [result of view transform]
- Model transform and view transform are often concatenated for efficiency reasons.
coordinate systems

- **Model space (aka object space)**
  - Being in model space means that a model has not been transformed at all.
  - A model can be associated with a *model transform* to position and orient it.
  - Several model transforms associated with one model allow for multiple instances without geometry replication.
• **World space**

• After the model transform has been applied to the model, it is located in world space.

• Model transform changes vertices and normals of the model.

• World space is unique: After the models have been transformed by their respective model transforms, all models exist in this same space.

right-hand coordinate system
coordinate systems

- Camera space
  - Virtual camera has a location in world space and a direction.
  - The view transform places the camera at the origin and aims it to look in the direction of the negative z-axis, with the y-axis pointing upwards and the x-axis pointing right.
  - All models are transformed with the view transform to facilitate projection and clipping.
let's colour the triangle
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f, -1.0f, 0.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);

    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
so let’s do some 3D drawing
start with framework from last example
void myDisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glVertex3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 1.0f, -1.0f, 0.0f);
    glVertex3f(-1.0f, -1.0f, 0.0f);
    glEnd();

    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    SDL_GL_SwapBuffers();
}
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_QUADS);
    // front
    glColor3f(0, 1, 0);
    glVertex3f(-1, 0, 1);
    glVertex3f(-1, 2, 1);
    glVertex3f(1, 2, 1);
    glVertex3f(1, 0, 1);
    glEnd();

    SDL_GL_SwapBuffers();
}
compile and run
void myinit(int width, int height)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (float)width/(float)height, 0.1, 100.0);
    gluLookAt(0.0, 2.0, 8.0, // eye
               0.0, 2.0, -1.0, // center
               0.0, 1.0, 0.0); // up
    glMatrixMode(GL_MODELVIEW);
}
drawing the remaining quads is trivial and left as an exercise to the student
drawing the other quads

// back
glVertex3f(-1, 0, -1);
glVertex3f( 1, 0, -1);
glVertex3f( 1, 2, -1);
glVertex3f(-1, 2, -1);
glVertex3f(-1, 2, -1);

// left
glVertex3f(-1, 0,  1);
glVertex3f(-1, 2,  1);
glVertex3f(-1, 2, -1);
glVertex3f(-1, 0, -1);
glVertex3f(-1, 0, -1);

// right
glVertex3f(1, 0,  1);
glVertex3f(1, 0, -1);
glVertex3f(1, 0, -1);
glVertex3f(1, 2, -1);
glVertex3f(1, 2,  1);
not much different, I’m afraid
just a question of perspective
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glRotatef(rotation, 0, 1, 0);

    glBegin(GL_QUADS);
    // front
    glColor3f(0, 1, 0);
    glVertex3f(-1, 0, 1);
    glVertex3f(-1, 2, 1);
    glVertex3f(1, 2, 1);
    glVertex3f(1, 0, 1);
    glEnd();

    SDL_GL_SwapBuffers();
}
rotating the scene

// in main()

while (!done) {
    mydisplay();
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT) done = true;
        if (event.type == SDL_KEYDOWN) {
            switch(event.key.keysym.sym) {
                case SDLK_ESCAPE:
                    done = true;
                    break;
                case SDLK_r:
                    rotation = (rotation + 5) % 360;
                    break;
            }
        }
    }
}
now for the pyramid...
drawing the pyramid

glBegin(GL_TRIANGLES);

    // front
    glColor3f(1, 1, 0);
    glVertex3f(-1, 2, 1);
    glVertex3f(0, 4, 0);
    glVertex3f(1, 2, 1);

    // right
    glVertex3f(1, 2, 1);
    glVertex3f(1, 2, -1);
    glVertex3f(0, 4, 0);

    // back
    glVertex3f(1, 2, -1);
    glVertex3f(-1, 2, -1);
    glVertex3f(0, 4, 0);

    // left
    glVertex3f(-1, 2, 1);
    glVertex3f(0, 4, 0);
    glVertex3f(-1, 2, -1);

    glEnd();
a few words on 3D transformations
transformations overview

- OpenGL uses 4x4 matrices for modeling transformations.
  - Why not 3x3?
  - You don’t want to know... (But I will tell you anyway.)
- Convenience functions for many operations:
  - `glRotate*()`, `glTranslate*()`, `glScale*()`
- Effects of transformations can be localized
  - `glPushMatrix()`, `glPopMatrix()`
manipulating the matrix stack

- **glPushMatrix()**
  - push all matrices in the current stack (determined by `glMatrixMode()`) down one level (the topmost matrix is duplicated)

- **glPopMatrix()**
  - pop the top matrix off the stack. The second matrix from the top of the stack becomes top, the contents of the popped matrix are destroyed.
OpenGL modelview matrix

- 4x4 matrix
- OpenGL uses column vectors instead of row vectors
- Matrices in OpenGL are defined like this:

\[
M = \begin{pmatrix}
m_0 & m_4 & m_8 & m_{12} \\
m_1 & m_5 & m_9 & m_{13} \\
m_2 & m_6 & m_{10} & m_{14} \\
m_3 & m_7 & m_{11} & m_{15}
\end{pmatrix}
\]
model transformations in OpenGL

- 3 modeling transformations
  - `glTranslate*()`
  - `glRotate*()`
  - `glScale*()`
- Multiply a proper matrix for transform/rotate/scale to the current matrix and load the resulting matrix as current matrix.
maths alert
\( \text{glScalef}(a, b, c) \)

- \( x_1 = ax_0;\ y_1 = by_0;\ z_1 = cz_0 \)

- How can we write this in matrix form?

\[
\begin{bmatrix}
  x_1 \\
  y_1 \\
  z_1
\end{bmatrix}
= \begin{bmatrix}
  a & 0 & 0 \\
  0 & b & 0 \\
  0 & 0 & c
\end{bmatrix}
\cdot
\begin{bmatrix}
  x_0 \\
  y_0 \\
  z_0
\end{bmatrix}
= \begin{bmatrix}
  ax_0 \\
  by_0 \\
  cz_0
\end{bmatrix}
\]

- Thus the scaling matrix is

\[
S = \begin{bmatrix}
  a & 0 & 0 \\
  0 & b & 0 \\
  0 & 0 & c
\end{bmatrix}
\]
Similarly for rotation we have:

- \( \text{glRotatef}(a, x, y, z) \)

- \( \text{glRotatef}(a, 1, 0, 0) \):
  \[
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & \cos a & -\sin a \\
  0 & \sin a & \cos a
  \end{bmatrix}
  \]

- \( \text{glRotatef}(a, 0, 1, 0) \):
  \[
  \begin{bmatrix}
  \cos a & 0 & \sin a \\
  0 & 1 & 0 \\
  -\sin a & 0 & \cos a
  \end{bmatrix}
  \]

- \( \text{glRotatef}(a, 0, 0, 1) \):
  \[
  \begin{bmatrix}
  \cos a & -\sin a & 0 \\
  \sin a & \cos a & 0 \\
  0 & 0 & 1
  \end{bmatrix}
  \]
• \texttt{glTranslatef(x,y,z)}

• How is a translation defined?

• \[ x_1 = x_0 + x \]
  \[ y_1 = y_0 + y \]
  \[ z_1 = z_0 + z \]

!! This is a problem!!

There is no way to represent this as a multiplication of 3x3 matrices.
• Where there’s a will, there’s a workaround.

• Use 4x4 matrices!

\[
T = \begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

• This actually gives us the correct results:

\[
\begin{bmatrix}
x_1 \\
y_1 \\
z_1 \\
1 \\
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \cdot \begin{bmatrix}
x_0 \\
y_0 \\
z_0 \\
1 \\
\end{bmatrix} = \begin{bmatrix}
x_0+x \\
y_0+y \\
z_0+z \\
1 \\
\end{bmatrix}
\]

• \texttt{glTranslatef(x,y,z)}
you can open your eyes again
order of transformations

- Matrix multiplication is not commutative.
- The order of operations is important!
- Example: Rotation and translation
  - rotate first, than translate
  - translate first, than rotate
glTranslatef( 0.00 , 0.00 , 0.00 );

glRotatef( 0.0 , 0.00 , 1.00 , 0.00 );

setScalef( 1.00 , 1.00 , 1.00 );

glBegin( . . . );

. . .

Click on the arguments and move the mouse to modify values.
This tutorial is based on my CG introduction course at the FH Technikum Wien.

It makes heavy use of the awesome OpenGL tutor apps by Nate Robins
http://www.xmission.com/~nate/tutors.html

Feel free to contact me with questions and comments at kyrah@kyrah.net.

These slides and the code examples are available at http://kyrah.net/scratch/opengl