OpenGL Application’s architecture

Glew extension library loader (3rd party)
Freeglut GLUT Utility ToolKit, system independent (3rd party) (GLFW is a more minimalistic Freeglut alternative)
GL OpenGL primitives
glm OpenGL Mathematics includes GLSL-like math
includes465 *.hpp files with helper functions for Comp 465 projects

on-line glut manual // many glut features are "compatibility mode"
API Description

State Machine

Functions set current state variables: polygon mode, culling

```c
 glEnable(GL_CULL_FACE)
 glCullFace(GL_BACK)
```

Function naming convention:

```c
<library> gl // core
<method name>
 [ <number of arguments> ]*
 [ <type of arguments> ]* {s,i,f,d,v}

glBindBuffer( GL_ARRAY_BUFFER, buffer );

glUniformMatrix4fv( MVP, 1, GL_FALSE,
 glm::value_ptr(modelViewProjectionMatrix) );
```
### Programming Languages

<table>
<thead>
<tr>
<th>Device</th>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>C++</td>
<td>Application specific OOP code classes for visual objects</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>glut function calls OpenGL function calls</td>
</tr>
<tr>
<td>GPU</td>
<td>glsl</td>
<td>vertex, shader manipulation</td>
</tr>
</tbody>
</table>

Everything is **not** an object

C++ is a hybrid OOP (procedural and OO). It can have methods (like main) outside of any class definition.

Application specific classes have methods that will make OpenGL calls. Class `Object3D`'s draw function could call `glDrawArrays(...)`

Do not attempt to write OpenGL/glut function handlers in classes.
OpenGL app's structure

- **User**
  - System (timer), keyboard, mouse events

  // C++ C code: glut, glm, OpenGL functions

  - glut*window (...) // interface to host OS
  - glutDisplayFunc (display) // draw fn()
  - glutReshapeFunc (resize) // resize fn()
  - glut*func(...) ➔ app fn(...) // handle events

  - glewInit() // enable extension handler
  - init(...) // create and load models ➔ buffers
    // load, compile, ... glsl programs
  - // create, update, M, V, P matrices

- **Application program**
  - glutMainLoop() // respond to events, render

  - glutPostRedisplay() ➔ glutSwapBuffer() ➔

- **Shader program**
  - vertex.glsl ➔ fragment.glsl

- **Display**
  - refresh GPU frame buffer
# define and # include

File: includes465/include465.hpp

/*
...
Contains several cpp defined constants for project builds.
All *465 constants start and end with 2 underscores.

__Linux__    // Ubuntu or other distro
__Mac__      // Mac OSX
__MinGW__    // Windows, Minimalist Gnu for Windows
__Windows__  // Windows, Visual Studio 201?)
__PI__       // Define for PI = glm::pi<float>(); value
...
*/

// defines and includes

# define __INCLUDES465__
# define GLM_FORCE_PURE     // for latest version of GLM
# define BUFFER_OFFSET(x)   ((const GLvoid *) (x))
# define MAX_INFO_LOG_SIZE

# include <stdio.h>    // My examples use printf,
# include <stdlib.h>
# include <sys/stat.h>
// OS specific includes
# ifndef __Linux__
# include <GL/glew.h>
# include <GL/freeglut.h>
# endif

# ifndef __Mac__
# include <GL/glew.h>
# include <GLUT/glut.h>
# endif

# ifndef __MinGW__
// put stuff for minGW here, __Linux__ might work
# endif

# ifndef __Windows__
# include <Windows.h>
# include <GL/glew.h>
# include <GL/freeglut.h>
# endif
// include the glm shader-like math library
#define GLM_FORCE_RADIANS // use radians not angles
#define GLM_MESSAGES    // compiler messages
#include <glm/glm.hpp>
#include <glm/gtc/constants.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtx/quaternion.hpp>

// print matrices and vectors, ...
#include "../includes465/glmUtils465.hpp"

// load vertex and fragment shaders
#include "../includes465/shader465.hpp"

// load AC3D *.tri model
#include "../includes465/triModel465.hpp"

ifndef __PI__ // set constant for PI, convience
#define __PI__
const float PI = glm::pi<float>();
endif
viewTriModel example

viewTriModel.cpp structure is typical of my examples

/*
Name and description of example
*/
// define your target operating system: ...
#define __Windows__
#include "../includes465/include465.hpp"

// Constant and variable declarations, can take many lines

void init (void) {
    // load *.tri models, could be many
    // load and bind shader programs
    // generate and bind buffers, attributes, uniforms
    // set cameras, enable depth testing, set background
}

void reshape(int width, int height) {
    // set viewports and projection matrix
}
void display(void) { // draw a frame
    // clear the viewport
    // set matrices: model, view, projection
    // draw the models
    // update fps values and window title
    glutSwapBuffers(); // display the new color buffer
}

// handle glut idle and fixed interval timer events
// for animation
void update(void){...}
void intervalTimer (int i) { ... }

// handle user keyboard events
void keyboard (unsigned char key, int x, int y) {...}
int main(int argc, char* argv[]) {
    // init glut, create window
    // init glew
    init(); // initialize application
    // set glut event callback functions
    glutMainLoop(); // run visual simulation
    return 0;
}

Briefly describe / walkthrough viewTriModel.cpp
Programmable Pipeline

client (application) \(\rightarrow\) display server

The application runs in the computer’s space and makes drawing requests to the display server (GPU)

Simplified programmable shader pipelines

- **program**
  - vertex shader
  - primitive assembly
  - fragment shader
  - frame buffer

- scale, translate, rotate vertices
- covert to view coordinates
- clip to viewing volume
- Determine lights' contribution to each pixel
- set pixel's color or texture
GLUT Callbacks

Callbacks are a indirect function calls ➔ event driven “on demand” apps
Connect user code with pre-built library routines.

- library has a callback resource (*entry in array...*) for user behavior
- user writes function, “registers / adds” function to callback resource
- executing program can call user function from library function

```
compiled library code
...
typedef struct {
    void (* fn)(...);
    ...; } Callback;
Callback cb[n];
...
void addCallback(
    void (* fn) (...), int cbId){...}
...
void leftButtonPress {} {
    cb[LBN]-> fn(...);
}
```

```
user’s code
...
void myPress(...) {...}
...
int main () {
    addCallback(myPress, LBN);
    ...
}
```

```
left button press
system
```

```
library
```

```
application
user’s function
```
GLUT Vs. openGL coordinate systems

Freeglut uses a window manager's coordinate system.

openGL uses a right handed coordinate system.

Freeglut mouse drag (x,y) values are in the Freeglut coordinate system....

https://www.opengl.org/sdk/docs/man/html/glViewport.xhtml

Viewport

    glViewport(x, y, width, height);  // origin, offsets

Initially the size of the window created.
Place in resize function to adjust accordingly.
int glutCreateWindow(char *name);

The initial window is at -1, -1 with a size of 300 by 300.

void glutInitWindowSize(int width, int height);

void glutInitWindowPosition(int x, int y);

void glutSetWindowTitle(char *name);

void glutSetIconTitle(char *name);

void glutFullScreen(void);

void glutSetCursor(int cursor);
GLUT display

`glutPostRedisplay()` explicitly calls the rendering function set with `glutDisplayFunc(...)`.  

`glutSwapBuffers()` causes double buffered output to be displayed -- usually last stmt in rendering function.  
the rendering function is also called on window changes after the call to the resize function set with `glutReshapeFunc(...)`.  
the render function has (int width, int height) args.

`glutTimerFunc(...)` sets the timer function to be called after an interval (msec) occurs.  A user specified int value can be passed to the timer function. (animation)

`glutIdleFunc(...)` sets the idle function to be called when there are no other events pending. (background sensor reading...)
GLUT keyboard

X and Y arguments are mouse position in window coordinates

```c
// set handler
glutKeyboardFunc(void (*func)(
   (unsigned char key, int x, int y))); // set handler

// define handler
void keyboardFuncName(unsigned char key, int x, int y) {
   if (key == 'q' || key == 'Q') exit();
}

// set handler
glutSpecialFunc(void (*func)(
   (int key, int x, int y));

// define handler
void specialFuncName(int key, int x, int y) {
   if (key == GLUT_KEY_F1) // do something;
      if (key == GLUT_KEY_UP &&
         glutGetModifiers() == GLUT_ACTIVE_CTRL) // pitch
      ...
}
```
Mouse movement w/ no buttons pressed is passive motion.

Need to keep old values of mouse position: say, \( mtX, mtY \)

Then relative mouse movement is:

\[
x - mtX \quad \text{and} \quad mtY - y
\]

// set handler
glutMotionFunc (void (*func)(int x, int y));

// define handler
void motionFuncName(int x, int y) {...}

// set handler
glutPassiveMotionFunc (void (*func)(int x, int y));

// define handler
void passiveMotionFuncName(int x, int y) {...}
Simplified view of the GeForce 6 Series architecture, used as a graphics pipeline.

Host
- CPU, memory
- OpenGL graphics app

GPU
- vertex arrays
- buffers, locations
- vertex attribute ptr
- uniform ptr
- shader program
VAO  vertex array object

OpenGL used int (GLuint) indexes into tables to "name" resources

Generate vertex array objects (VAO)

```c
void glGenVertexArrays(GLsizei nNames, GLuint * arrays)

nNames     number of unused "index" names needed
arrays     "table" to hold values of "names"
```

Create storage for vertex array (instantiation)

```c
void glBindVertexArray(GLuint array)

when array == 0    reset to default value
when array != 0    new VAO, VAO is active (current)
```

After VAO is "bound" to buffer, the VAO can be deleted. With small / simple OpenGL apps this step may not be done (memory is not needed).

```c
void glDeleteVertexArrays(GLsizei n, GLuint * arrays)
```
Buffers (VBOs) vertex buffer objects

Store VAOs in buffer objects, "manages" by "currently bound" or active VAO.

Create names for VBOs

```c
void glGenBuffers(GLsizei n, GLuint * buffers)
```

Create storage for VBOs (instantiation)

```c
void GLBindBuffer( GLenum bufferType, GLunit buffer)
bufferType GL_ARRAY_BUFFER, ... // 8 buffer types
buffer == 0 stop using this buffer
buffer != 0 1\textsuperscript{st} time create buffer, assign "current" name, other time. make "current" buffer
```

Can be deleted (usually need the buffer)

```c
void glDeleteBuffers(GLsizei n, const GLuint * buffers)
```
Model data can be specified in app (pyramid.cpp), loaded from a model file (cubeTriModel.cpp, 3DModeling notes), or generated procedurally.

Allocate storage units (bytes) for storing data

```c
void glBufferData( GLenum bufferType, GLsizeiptr size, const GLvoid * data, GLenum usage)
```

- **bufferType**: GL_ARRAY_BUFFER // other types
- **size**: how much memory ( n * sizeof(dataType) )
- **data**: base address of data to be copied
  - if NULL, space reserved to be filled later
- **usage**: how to read / write data after allocation
  - GL_STATIC_DRAW

Initialize part of a buffer // glBufferData's data == NULL

```c
void glBufferSubData(GLenum bufferType, GLintptr offset, GLsizeiptr size, const GLvoid * data)
```

- **offset**: base + offset byte position to store data
Dynamic application values {position or orientation changes, user input, ...} may need to be available to the shader programs.

Done after shader program has been established ...

Associate a shader attribute location with a vertex attribute array

```c
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, GLvoid * pointer)
```

- **index** shader variable position
- **size** number of components in value
- **normalized**
- **stride** interval between values in buffer
- **pointer** offset into buffer
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, GLvoid * pointer)

element from pyramid.cpp  // vertex is vector 4 (x, y, z, h)

GLuint vPosition = glGetUniformLocation(shaderProgram, "vPosition");

glEnableVertexAttribArray(vPosition);

glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
GLSL program uniform type is for variables that are passed from the application program to the shader. Matrices are a common uniform type passed to the shader. More on GLSL later.

Application program needs to get the location of the uniform variable in the shader

```c
Gluint glGetUniformLocation(GLuint shaderProgram, const char * variableName);
```

Application sets the value of the uniform

```c
void glUniform {[1,2,3,4}{f,d,i,u,i} (GLuint location, TYPE value)
void glUniformMatrix{2,3,4}{f,d} v (GLuint location, GLsizei count, GLboolean transpose, const GLfloat * values)
```
Example from pyramid.cpp

GLuint MVP; // ModelViewProjection handle
glm::mat4 projectionMatrix; // set in display
glm::mat4 modelViewMatrix; // set in display()
glm::mat4 modelViewProjectionMatrix; // set in display()

MVP = glGetUniformLocation(shaderProgram,
    "ModelViewProject"); // init()

// update for vertex shader in display()
modelViewProjectionMatrix = projectionMatrix * 
    modelViewMatrix;
glUniformMatrix4fv(MVP, 1, GL_FALSE,
    glm::value_ptr(modelViewProjectionMatrix));
vertex and fragment shaders

Shaders from pyramid example w/o comments

pyramidVertex.glsl

```glsl
# version 330 core

in vec4 vPosition;
in vec4 vColor;

uniform mat4 ModelViewProject;
out vec4 color;

void main() {
    color = vColor;
    gl_Position = ModelViewProject * vPosition;
}
```

pyramidFragment.glsl

```glsl
# version 330 core

in vec4 color;

out vec4 fragColor;

void main() { fragColor = color; }
```
Conceptual Orientation & Translation matrix

Successive matrix multiplications, transformations, generate a concatenated ModelView matrix.

Post matrix multiplication // right to left

This matrix contains orientation and position for the current "reference frame"

OpenGL matrices contain 16 GLdouble or GLfloat values

GLfloat m[16]; // or GLdouble,

GLM matrices

glm::mat4 m; // float or double


conceptual matrices revisited in Transformations notes
Model View Projection

scene (a frame) \( \leftrightarrow \) [Projection] \* [View] \* [Model] \* \{ vertex \}

Consider a vertex "v" at \((10, 10, 0, 1)\)
that is being viewed with a camera
at \((0, 0, 100)\)
looking at \((0, 0, 0)\)
up orientation \((0, 1, 0)\)

Projection as identity (simplification)

View matrix

\[
\begin{array}{ccc|c}
\text{Right} & \text{Up} & \text{At} & \text{Pos} \\
\hline
X & 1.00 & -0.00 & 0.00 & -0.00 \\
Y & 0.00 & 1.00 & 0.00 & -0.00 \\
Z & -0.00 & -0.00 & 1.00 & -100.00 \\
W & 0.00 & 0.00 & 0.00 & 1.00 \\
\end{array}
\]

No M or P transforms yet

\[
\text{MVP} = P \* V \* M
\]

\[
v' = \text{MVP} \* v
\]

\[
v' = (10, 10, -100, 1)
\]
Move the camera // look from right at (100, 0, 0)
looking at (0, 0, 0)
up orientation (0, 1, 0)

View matrix

<table>
<thead>
<tr>
<th>Right</th>
<th>Up</th>
<th>At</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0.00</td>
<td>0.00</td>
<td>-1.00</td>
</tr>
<tr>
<td>Y</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Z</td>
<td>1.00</td>
<td>-0.00</td>
<td>-0.00</td>
</tr>
<tr>
<td>W</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

No model or projection transforms; M and P are identity matrices

Model View Projection matrix

<table>
<thead>
<tr>
<th>Right</th>
<th>Up</th>
<th>At</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0.00</td>
<td>0.00</td>
<td>-1.00</td>
</tr>
<tr>
<td>Y</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Z</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>W</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

\[ \text{MVP} = P \times V \times M \]

\[ \mathbf{v}' = \text{MVP} \times \mathbf{v} \]

\[ \mathbf{v}' = (0, 10, -90, 1) \]

see ModelViewProject.cpp
GLM is an open source include only math library with a syntax similar to GLSL math (with some odd syntax due to its no namespace design).

GLM provides

- **vectors**
  - vec2, dvec2, vec3, dvec3, vec4, dvec4  // float, double

- **normalize vectors**
  - vec3 = normalize(vec3)

- **matrices**
  - mat3, dmat3, mat4, dmat4

- **viewMatrices**
  - lookAt(vec3 eye, vec3 at, vec3 up)

- **perspective**
  - perspective(FovY, aspectRatio, near, far)

- **orthographic**
  - ortho(left, right, bottom, top, near, far)

- **rotation**
  - rotate(mat4, radians, vec3 axis)

- **scale**
  - scale(mat4, vec3)

- **translation**
  - translate(mat4, vec3)

- **concatenation**
  - mat4 * mat4

- **products**
  - float = dot(vec3, vec3), vec3 = cross(vec3, vec3)

- **pointer to memory**
  - value_ptr(...)

**```cpp**
vec3 = normalize(vec3)
```
```cpp
#include "../includes465/include465.hpp"

glm::mat4 projectionMatrix;
glm::mat4 modelViewMatrix;
glm::vec3 eye, at, up; // vectors and values for lookAt

// rotation variables
GLfloat rotateRadian = 0.0f;
glm::mat4 identity(1.0f); // initialized identity matrix

glm::mat4 rotation;

...  

// reshape function
aspectRatio = (GLfloat) width / (GLfloat) height;
projectionMatrix = glm::perspective(glm::radians(60.0f), aspectRatio, 1.0f, 1000.0f);

...  

// display function
rotation = glm::rotate(identity, rotateRadian, glm::vec3(0, 1, 0)); // yaw rotation
modelViewMatrix = glm::lookAt(eye, at, up) * rotation;
```
glm has "compatibility mode" like transformations methods that use float or double arguments to create transformation matrices

translate – movement along axes change position, not orientation

mat4 TranslateMatrix = translate(  
    mat4 initialMatrix, vec3 translateVector);

glm::mat4 TranslateMatrix = glm::translate(  
glm::mat4(1.0f), glm::vec3(-5, -20, 0);

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>Up</td>
<td>At</td>
<td>Pos</td>
</tr>
<tr>
<td>X</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Y</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Z</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>W</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

v' = TranslateMatrix * vertex  // vertex = (10, 10, 0, 1)  
v' == (5, -10, 0)
rotate – movement about an axis, change position and orientation

glm can use angle degrees like OpenGL, or radians, like GLSL

We'll use radians.

```cpp
mat4 RotationMatrix = rotate(
    mat4 initialMatrix, float radian, vec3 axis);

glm::mat4 RotationMatrix = glm::rotate(
    glm::mat4(1.0f), glm::radians(45.0),
    glm::vec3(0, 1, 0); // axis of rotation, here Y
```

Rotation matrix

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>Up</td>
<td>At</td>
<td>Pos</td>
</tr>
<tr>
<td>X</td>
<td>0.71</td>
<td>0.00</td>
<td>0.71</td>
</tr>
<tr>
<td>Y</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Z</td>
<td>-0.71</td>
<td>0.00</td>
<td>0.71</td>
</tr>
<tr>
<td>W</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

There is also a scale transformation, that changes positions with a multiplier.

\[ v' = RotationMatrix \times \text{vertex} \]
\[ v' = (7.07, 10.00, -7.07, 1.00) \]
Big Bang !!!!

In your initialization load models centered at the origin

Create models with their center at the origin
   orient model’s forward towards \(-Z\) (looking into screen)
Set your [View]

In your user interaction functions set changes to the [View]

In your display apply transformations to [Model] before drawing

Order matters!
   \([\text{Rotate}] \times [\text{Translate}] \neq [\text{Translate}] \times [\text{Rotate}]\)

When loading at origin
   rotation about self-center \([\text{Translate}] \times [\text{Rotate}]\)
   orbital rotation \([\text{Rotate}] \times [\text{Translate}]\)

<< see Robin's Tutors transformation.exe >>
Animation

Uses the Freeglut Idle callback function(or timer) to set values for transformation matrices and/or render states for objects

```c
// void glutTimerFunc (void (*func) (void))

void update(void) {
    // set next timer event
    glutTimerFunc(40, update, 1);
    // set up animation values
    glutPostRedisplay();
}

... 

void render (GLenum mode){
    // draw
}

... 

void main () {
    ... 
    glutTimerFunc(40, update, 1);
}
```
Projection

Orthographic project is a parallel projection of 3D into 2D (disregard the z axis values...). Useful for CAD... see text examples.

```cpp
glm::ortho(left, right, bottom, top, near, far)
```

```cpp
glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1000.0f);
```

Perspective projection defines a “vanishing point” on +Z and scales point position by their distance from the viewer.

```cpp
glm::perspective(FieldOfViewAngleY, aspectRatio, near, far);
```

```cpp
glm::perspective(glm::radians(45.0f), aspectRatio, 1.0f, 1000.0f);
```

Projection defines a viewing volume or frustum:

- top
- left
- right
- bottom
- near
- far
Define a “sight” vector from the eye to the center of the scene.
Define an “up” vector.

```cpp
mat4 glm::lookAt(
    vec3 eye position,
    vec3 looking at position,
    vec3 up orientation);
```

```cpp
glm::vec3 eye  = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 at   = glm::vec3(0.0f, 0.0f, 0.0f);
glm::vec3 up   = glm::vec3(0.0f, 1.0f, 0.0f);
glm::mat4 View = glm::lookAt(eye, at, up);
```

Can’t have the sight and up vector be the same.
Won’t see anything. Lighting incorrect (later)

```cpp
glm::lookAt( 0, 0, 0,
              0, 10, 0,
              0, 100, 0);
<< see Robin's Tutors projection >>
```
Surface normal

Dot and cross products are used in lighting calculations usually in the fragment shader.

Surface normal is the cross product of two vectors defining a surface (iff triangle).

Order of arguments is important.

cross(v1, v2) \neq cross(v2, v1)  \ // \ direction \ is \ flipped

```cpp
glm::vec3 normal;
glm::vec3 point[3];  \ // 3 vertices of a triangle
normal = glm::normalize(
    glm::cross( point[1] - point[0], point[2] - point[0] ));
\ // \ cross (left\Vector, \ right \ Vector)
```
C++ classes and C application

// Separate Object Oriented from C / OpenGL

class Cube {
    ...  
    public void draw() {...}
    }

Cube * aCube;  // pointer (reference) to a Cube

public void draw() { ...
    aCube -> draw();
    }

// Program entry
int main(int argc, char *argv[]) {
    aCube = new Cube();   // create objects
    ... // create glut display and window
    glutDisplayFunc(draw);  // set callback functions
    ...
    glutMainLoop();
    ...
}

C++ methods have hidden first argument (this, ...)

Single model used to draw 1,000 Shape3D objects.

Each Shape3D object provides its own model matrix

```c
// Shapes
const int nShapes = 1000;
Shape3D * shape[nShapes];

model matrix set in
shape[i] -> updateDraw()

projection matrix set in reshape()

view matrix set in keyboard()

ModelViewProjectionMatrix set in
display()
```

C for OpenGL application file
C++ for “scene graph”
FPS by Number of Shapes

<table>
<thead>
<tr>
<th>shapes</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6910</td>
</tr>
<tr>
<td>100</td>
<td>1050</td>
</tr>
<tr>
<td>1000</td>
<td>111</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
</tr>
</tbody>
</table>

VS 2012
32 bit
gforce gt530
ManyModels

```c
#include "includes465/include465.h"

- nModels : const int
- nSolids : const int
// shaderFileNames : char *
modelFile : char * [nModel]
VAO : GLuint [nModels]
VBO : GLuint [nModels]
// shader variable handles
nVertices : int [nModels]l
// other app variables
modelIBR : int [nModels]
solid : Solid3D * [nSolids]
// various matrices : mat4
+ main(argc : int, argv : char **) : int
+ init() : void // initialize app
// app fnos()
...
// glut callback handlers
...
+ display() : void
+ keyboard(key : unsigned char , x : int , y : int) : void
+ reshape(width : int , height : int)
```

```
<< class >>
Obj3D

# name : char[25]
# scaleMatrix : mat4
# rotationMatrix : mat4
# translationMatrix : mat4
+ Obj3D (aName : char[25] // init matrices w/ identity
c + getTranslationMatrix() : mat4
c + getRotationMatrix() : mat4
```

```
<< class >>
Solid3D

- rotationAxis : vec3
- radians : float
- orbital : bool
+ Solid3D(char * name : char *, size : float, translate : vec3,
  rotRadians : float, rotAxis : vec3, sOrbital : bool)
+ getModelMatrix() : mat4
+ setTranslation(translate : vec3) : void
+ setRotation(rotRadians : float , rotAxis : vec3) : void
```
scene initialization in application file

const int X = 0, Y = 1, Z = 2, START = 0, STOP = 1;
// constants for models: file names, vertex count, ...
const int nModels = 3;  // number of models in this scene
const int nSolids = 8;  // number of Solid3D instances to display
char * modelFile[nModels] = {
    "axes-r100.tri",
    "obelisk-10-20-10.tri",
    "spaceship-bs100.tri"};
float modelBR[nModels];  // model's bounding radius
float scaleValue[nModels];  // model's scaling "size" value
const int nVertices[nModels] = { 120 * 3, 14 * 3, 996 * 3};

Solid3D * solid[nSolids];  // create the solids

// {START, STOP} 1 axis {0}, 4 obelisks {1..4}, 3 ships {5..7}
int instance[nModels][2] = {
    {0, 1},
    {1, 5},
    {5, 8}};
float instanceTranslate[nSolids][3] = {
    {0, 0, 0},
    {0, 0, 3000},
    {0, 2000, 0},
    {0, 500, 0},
    {0, 0, 1000},
    {0, 1414, 1414},
    {0, -1414, 1414},
    {750, 0, 0}};
float instanceRadian[nSolids] = {
    0.0f,
    0.01f,
    0.03f,
    0.01f,
    0.03f,
    0.03f,
    0.02f,
    0.01f};
float instanceAxis[nSolids][3] = {
    {0, 1, 0},
    {0, 1, 0},
    {1, 0, 0},
    {0, 1, 0},
    {1, 0, 0},
    {0, 0, 1},
    {1, 0, 0},
    {0, 1, 0}};
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // for each model draw its instances -- from
    // solid[start] to solid[stop]
    for (int m = 0; m < nModels; m++) {
        glBindVertexArray(VAO[m]); // set model for its instances
        for (int i = instance[m][START]; i < instance[m][STOP]; i++) {
            modelMatrix = solid[i]->getModelMatrix();
            ModelViewProjectionMatrix = projectionMatrix * viewMatrix * modelMatrix;
            glUniformMatrix4fv(MVP, 1, GL_FALSE,
                glm::value_ptr(ModelViewProjectionMatrix));
            glDrawArrays(GL_TRIANGLES, 0, nVertices[m]);
        } // instances
    } // models
    glutSwapBuffers();
    frameCount++;
    // update fps etc
    ...
}
Models | vertices
--- | ---
axes | 360
obelisk | 42
spaceship | 2988

Solid3D | size
--- | ---
"axes 1" | 1,000
"obelisk 1" | 300
"obelisk 2" | 200
"obelisk 3" | 50
"obelisk 4" | 50
"ship 1" | 300
"ship 2" | 200
"ship 3" | 100

Obelisk 3 and 4 are used as "ticks", they mark 500 on Y and 1000 on Z

includes465/triModel465.hpp has loadModelBuffer(....) function dynamically allocates arrays for model's vertex, color and normal data

see ManyModelsStatic project for a simpler example in code