Project 2
Introduction to Computer Graphics

Andrzej Szymczak

October 25, 2013

Rendering the mesh

- Use the triangle soup mode (easiest to do)
- Remember you have to compute and specify normals for each vertex in order to get illumination working
- Normals are not provided with the input file
- Triangles are oriented consistently, so back-face culling should work
  - If back-face culling causes the back faces to be displayed by your implementation, just reorient your triangles (swap any pair of vertices, together with their attributes)

Mesh: Input file

- First two entries: triangle count T and vertex count V
- Triangle table (T triples of integers), sequence of triples of integer vertex IDs (in the range $0 \ldots V-1$) that define a triangle
- Vertex table coordinates of consecutive vertices
- Example: a tetrahedron

```
4 4
0 1 2
2 1 3
2 3 0
0 3 1
0.000000 0.000000 1.000000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 0.000000
```

Vertex normals

- Vertex normals are typically computed as an average of normals of incident triangles
  - The process described below is tailored for smooh, high-resolution models (small triangles)
  - For smaller models, using the true geometric triangle normals may make more sense
- Normal of a vertex v = sum of cross products computed for all incident triangles
  - An area-weighted average
- How to compute the normals efficiently?

Vertex normals

- Linear-time algorithm:
  - Allocate a vector $N[]$ of V normals (V=number of vertices), initialize all entries to zero
  - For each triangle $\Delta(a, b, c)$ do
    - Below, a,b and c are considered integer IDs of vertices or their coordinates depending on context
    - Compute normal of Δ, $N_\Delta = \vec{ab} \times \vec{ac}$
    - $N[a] += N_\Delta$
    - $N[b] += N_\Delta$
    - $N[c] += N_\Delta$
- Since triangles are oriented consistently, all cross products will point in a consistent direction (i.e. will either be all outward or all inward)
- Want to use outward normals

Sending the mesh data to GPU

- In modern OpenGL, the information you need for rendering is stored on the GPU
- Need to upload arrays containing vertex (or other) data into a *buffer*
- Buffer is basically an array residing in the GPU memory
- GPU can access information in a buffer an order of magnitude faster than the same information in the main memory (e.g. 200GB/s vs 5GB/s)

Constructing the buffers

- First, build arrays containing vertex coordinates and normal vector coordinates in the main memory
  - It would also be possible to use a single array
  - Here we use one per attribute for simplicity
  - In most cases, you want buffers describing the same 3D model to contain the same number of attributes
  - Attributes could be of different sizes

Constructing the buffers

```
NormalArray = new GLfloat[3*verts]; // 3 floats per vert
CoordArray = new GLfloat[3*verts]; // for both attr's
i := 0;
```

`for each triangle` $\Delta(a, b, c)$ `do:`

`. NormalArray[3*i+0] =` $N[a].x$`;`

`. NormalArray[3*i+1] =` $N[a].y$`;`

`. NormalArray[3*i+2] =` $N[a].z$`;`

`. CoordArray[3*i+0] :=` $v_a.x$`;`

`. CoordArray[3*i+1] :=` $v_a.y$`;`

`. CoordArray[3*i+2] :=` $v_a.z$`;`

`. ++i;`

`.` *do the same for vertex b...*

`. ++i;`

`.` *do the same for vertex c...*

`. ++i;`

`done`

Sending buffers to GPU

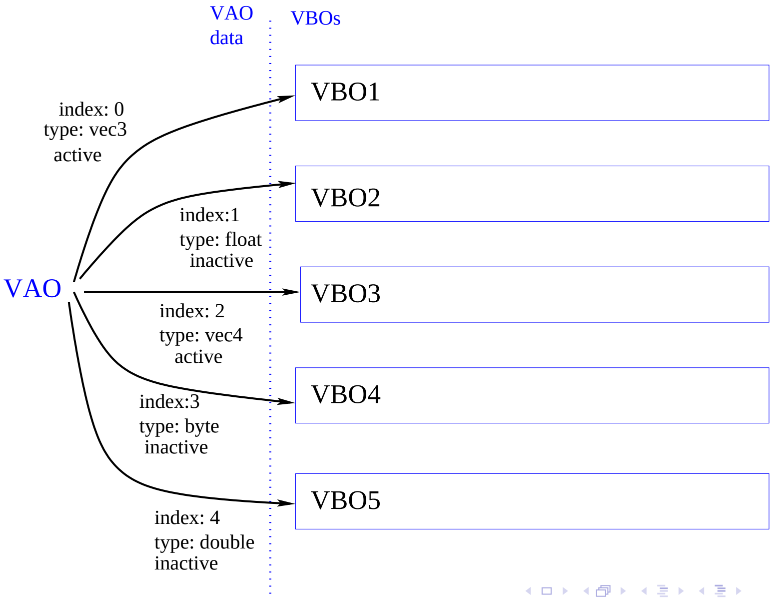
- Use the provided `Buffer` class for simplicity
- Remember:
  - A wrapper for an OpenGL Vertex Buffer Object (VBO)
  - No standard value/copy semantics implemented (usually no need for it)
  - Use pointers to `Buffer` objects in your code for best results
- `Buffer` Constructor
  - takes 3 arguments
    - number of *components* (basically, scalar values of the attribute) per entry; up to 4 allowed
    - number of entries
    - pointer to the array you want to put into the buffer; the code provides several overloaded versions to take care of different data types (note: no `double`/`GLdouble`!)
  - creates a buffer and sends array into it; stores the respective VBO information

```
Buffer *loc = new Buffer(3,verts,CoordArray);
Buffer *normal = new Buffer(3,verts,NormalArray);
```

Vertex Array Objects (VAOs)

- A vertex typically has several different attributes
- Need a mechanism to tell the system which buffer to use to generate *i*-th attribute
- This is what VAOs are for
- `VertexArray`: a wrapper class for VAOs
- Construct an empty VAO using the default constructor
  `myVAO = new VertexArray;`
- No copy/value semantics implemented - use pointers only
- Then, attach buffers to your VAO

VAOs and VBOs



Vertex Array Objects (VAOs)

- Attaching buffers to VAOs
- Use `attachAttribute` method

```
myVAO->attachAttribute(0,loc);
myVAO->attachAttribute(1,normal);
```

- If you use this VAO to generate vertices:
  - Attribute #0 will be looked up from the `loc` buffer (here: it will be location info)
  - Attribute #1 will be looked up from the `normal` buffer (for a given vertex, it will be the normal vector)
- Limitations of the OpenGL object wrappers
  - Don't use double precision attributes
  - Don't use matrix or array attributes
  - Don't mix floats with integer types (e.g. don't buffers with integer contents to specify values of floating point vertex attributes or the other way around)
  - `GLbyte`, `GLubyte`, `GLshort`, `GLushort`, `GLint`, `GLuint`, `GLfloat` attribute component types supported; in this project, `GLfloat` should be all you need

Vertex program

- C-like language (GLSL)
- Executed on the device (GPU)
- CPU–GPU communication needed for this to work!
- Input variables for the vertex shader: vertex attributes
  - Location qualifier provides the index information (this index has to match the index provided in the `attachAttribute` call)
  - For the VAO defined on previous slides, you would do something like this:

    ```
    layout(location=0) in vec3 coord;
    layout(location=1) in vec3 normal;
    ```
- Now the system will know to put attribute #0 into `coord` and attribute #1 into `normal`

Vertex program: gl_Position

- Vertex shader assigns the projected vertex coordinates to the built-in output variable `gl_Position` of type `vec4`
- This provides information needed in the rasterization stage
  - Don't do division by the homogenous coordinate in the shader!
  - Typically, what you do is something like this:
    `gl_Position = P * MV * coord;`
    where `P` and `MV` are the projection and modelview matrices of type `mat4` and `coord` is a `vec4` containing vertex locations in the model coordinates
  - This can be done differently e.g. because:
    - World coordinates are needed for illumination calculations; you apply `MV` only, store the result in a variable, use it to evaluate the Phong's formula and then apply `P`
    - Some components of the modelview matrix may be represented differently (e.g. translation using a `vec3` vector, scaling using a `float` scale factor or rotation using a `mat3` instead using a 'full' `mat4` modelview matrix).

Converting vectors

- From standard to homogenous
  - Converting from `vec3` to homogenous coordinates:

    ```
    ... vec3 coord;
    vec4 coordh = vec4(coord,1.0);
    ```
  - Similarly from `vec2` to homogenous coordinates (useful for flat shapes):

    ```
    ... vec2 coord;
    vec4 coordh = vec4(coord,0.0,1.0);
    ```
- Dropping or reordering coordinates
  - Swizzle operations
  - Examples: `coordh.xyz`, `coordh.x`, `rgba.rgb`, `coordh.zyx`, `coordh.wwww`, etc
  - for one coordinate, the result is of type `float`; otherwise, it's `vecN` , with `N` equal to 2, 3 or 4
  - Can be used as l-values if no two repeated coordinate names

Vertex program: user defined output variables

- You can send some number of attributes with the processed vertex
- These attributes are interpolated on rasterization stage and send with fragments to fragment processing stage
- Declare an output variable using a line that contains:
  - interpolation qualifier (`flat`, `noperspective`, `smooth`); smooth is the default and can be omitted; noperspective = linear, smooth = perspectively correct
  - key word `out`
  - type name (e.g. `float`, `int`, `vec3`, `vec4...`)
  - variable name; use the same name to declare the respective input to the fragment shader
- Examples of output variable declarations from the sample code
  - `out vec2 param;`
    This one uses the default (perspectively correct) interpolation
  - `flat out uint face_id;`
    ... and this one flat interpolation

Vertex program: user defined output variables

- Assign the values to all output variables in the shader's `main` function
- Other functions can be defined in GLSL code (probably no need to do that in out projects, but could be helpful for more complex shaders)
  - Functions may return values
  - Value-return convention: `in`, `out`, `inout` parameters

Fragment program

- Input variables need to match the output variables of the vertex shader
  - If the vertex shader has
    `flat out uint face_id;`
    then the fragment shader needs to have
    `flat in uint face_id;`
  - If the vertex shader has
    `out vec2 param;`
    then the fragment shader needs to have
    `in vec2 param;`
- In our projects, it is fine to declare a single output variable of type `vec3` and assign the final RGB values of the fragment to it in the `main()` function

Program objects

- The provided code makes dealing with them almost trivial
- Creation
  - Place the shader source code in two files (one for vertex, one for fragment shader)
  - Use the provided function `createProgram` to read, compile and link the shader code
  - Example from the sample code:

```
cube_program = createProgram("shaders/vsh_cube.glsl","shaders/fsh_cube.glsl");
```

  - Return value is of type `Program*`
  - **Don't use absolute paths!** or the grader will be mad...
- Turning on or off
  - `on` and `off` methods
  - `on` makes the program active, e.g. `cube_program->on();`
  - `off` turns it off (more precisely: an invalid program whose result is undefined)
  - No memory; don't turn on two programs at once
- Deletion: use the provided destructor:
  `delete cube_program;`

Program objects: uniform variables

- Uniform variables are technically not constant, but close to constant
  - they cannot be changed while a vertex stream is processed
  - ... but they can be altered (from the host code) between vertex streams
- Setting uniform variables
  - Use the `setUniform` method, for example
    `cube_program->setUniform("T",-0.8f,0.8f);`
    - Arguments: name of the variable followed by values
    - Several overloaded versions available; examples:
      ```
      p->setUniform("A3Dvector",1f,1f,1f);
      GLfloat v[4] = { 1.0, 2.0, 3.0, 1.0 };
      p->setUniform("A4Dvector",v);
      ```
    - The method should recognize how many components should be sent automatically, but it's best not to try to do anything that clearly does not make sense (e.g. send 4 numbers to a variable of type `vec2`)

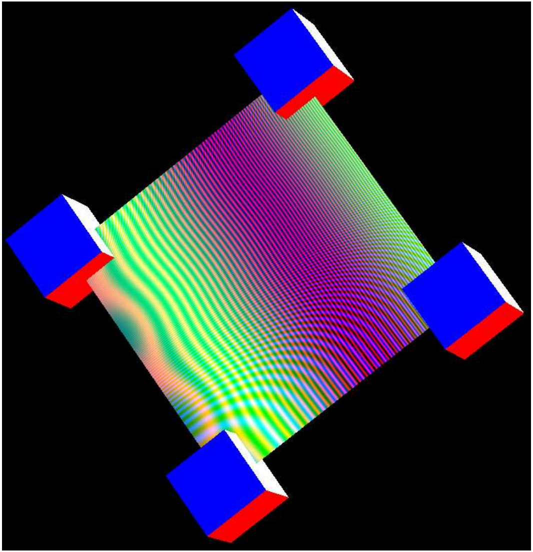
Program objects: uniform variables

- General remarks on uniform variables
  - Each program object has a separate set
  - Shaders within a program object can share uniform variables; for example, if you declare
    `uniform vec2 V;`
    in both vertex and fragment shader, you can use it in both shaders (the value is the same in both and a single call to `setUniform` can be used to set it)

A few useful built-in GLSL functions

- `float dot ( vec3 a, vec3 b )` computes the dot product of the arguments.
- `vec3 cross ( vec3 a, vec3 b )` computes cross product.
- Scalar by vector multiplication, vector addition, subtraction and negation are available and work like overloaded operators. For example, if you have two variables `w` and `v` of type `vec3`, you can do `-w+0.8*v`.
- `vec3 normalize ( vec3 a )` returns vector a normalized.
- Matrix by matrix multiplication and matrix by vector multiplication can be done by using overloaded $*$ operators. Examples:
  - For `M` of type `mat4`, i.e. 4×4 matrix, and a variable `v` of type `vec4`, you can do `M*v` to multiply them.
  - There are types and operators for matrices of other dimensions (up to 4) available too – you may find `mat3` useful for storing normal transformation (equivalently, the rotational part of the modelview matrix).

Sample code output



Example: the cube program from sample code

```
#version 420

layout (location=0) in vec3 modelCoord;
layout (location=1) in uint faceId;

flat out uint face_id;

uniform mat4 MV;
uniform mat4 P;
uniform vec2 T;

void main()
{
  face_id = faceId;
  vec4 worldCoord = MV *
     vec4(0.4*(modelCoord-vec3(0.5,0.5,0.5))
      +vec3(T,0),1.0);
  gl_Position = P * worldCoord;
}
```

```
#version 420

flat in uint face_id;

out vec3 fragcolor;

void main()
{
  switch(face_id)
  {
    case 0: fragcolor = vec3(1,0,0); break;
    case 1: fragcolor = vec3(0,1,0); break;
    case 2: fragcolor = vec3(0,0,1); break;
    case 3: fragcolor = vec3(1,1,0); break;
    case 4: fragcolor = vec3(1,1,1); break;
    case 5: fragcolor = vec3(1,0,1); break;
    default: fragcolor = vec3(.5,.5,.5); break;
  }
}
```

Example: the square program from sample code

```
#version 420

layout (location=0) in vec2 model_coord;

out vec2 param;

uniform mat4 MV;
uniform mat4 P;

void main()
{
  vec3 scaled_coords = vec3(0.8*model_coord,0);
  param = 0.5*(model_coord+vec2(1,1));
  gl_Position = P * MV * vec4(scaled_coords,1);
}
```

```
#version 420

in vec2 param;

out vec3 fragcolor;

void main()
{
  float x = 512.0*param.x;
  float y = 512.0*param.y;

  fragcolor = vec3(
       0.5+0.5*sin(sin(x/30.0)+y*y/700.0),
       0.5+0.5*sin(y/71.0),
       0.5+0.5*sin(x*x*x/120000.0+y*y/1700.0)
      );
}
```

Sending vertices into pipeline

- Two ways to do this
  - 'plain': just look up values of attributes of consecutive vertices from the consecutive entries in the buffers attached to the VAO
  - indexed: use an index buffer to provide offsets for attribute lookup from the VAO
- Quite often, indexed saves memory (in particular, when some vertices need to be sent multiple times)
- Code for the plain method:
  `myVAO->sendToPipeline(GL_TRIANGLES,first,num);`
  - First argument provides primitive setup control (`GL_TRIANGLES` means triangle soup; other values allowed: `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`, `GL_LINES`, `GL_LINE_STRIP`, `GL_POINTS`)
  - 2nd argument: starting index
  - 3rd argument: how many vertices to form
  - attributes are looked up from the buffers attached to `myVAO` using offsets `first`, `first+1`,`...first+num-1`

Sending vertices into pipeline: indexed

- Need an index buffer
- Use `IndexBuffer` class; as usual, use pointers only
- Construction: `myIx = new IndexBuffer(size,ptr);`
  - first argument: number of entries
  - second argument: pointer to an array containing the indices; must be of one of the unsigned types, `GLubyte`, `GLushort`, `GLuint`; of course, the array must be of size at least `size`
  - sends the data from the main memory to the GPU
- Code that creates the vertex stream:

```
myVAO->sendToPipelineIndexed(GL_TRIANGLES,myIx,first,num);
```

  - first argument: see previous slide
  - 2nd argument: pointer to `IndexBuffer` to be used
  - 3rd argument: index of the first vertex (into the index buffer)
  - 4th argument: number of vertices to be generated
  - Offsets used to generate vertex properties from buffers attached to the VAO: `myIx[first]`, `myIx[first+1],...myIx[first+num-1]`.

Sending vertices into the pipeline: example 1

```
GLfloat vertices[] = {
  -1, -1,
  -1,  1,
   1,  1,
   1, -1,
};
GLuint indices[] = {
  0, 1, 2,   // indices into the vertices of the first triangle
  0, 2, 3    // ... and second triangle (with consistent orientation)
};
.............
VertexArray *va_square = NULL;
Buffer *buf_square_vertices = NULL;
IndexBuffer *ix_square = NULL;
.............
buf_square_vertices = new Buffer(2,4,square_vertices);
ix_square = new IndexBuffer(6,square_indices);
va_square = new VertexArray;
va_square->attachAttribute(0,buf_square_vertices);
.............
va_square->sendToPipelineIndexed(GL_TRIANGLES,ix_square,0,6);
```

Example: the square programs

```
#version 420

layout (location=0) in vec2 model_coord;

out vec2 param;

uniform mat4 MV;
uniform mat4 P;

void main()
{
  vec3 scaled_coords = vec3(0.8*model_coord,0);
  param = 0.5*(model_coord+vec2(1,1));
  gl_Position = P * MV * vec4(scaled_coords,1);
}
```

```
#version 420

in vec2 param;

out vec3 fragcolor;

void main()
{
  float x = 512.0*param.x;
  float y = 512.0*param.y;

  fragcolor = vec3(
       0.5+0.5*sin(sin(x/30.0)+y*y/700.0),
       0.5+0.5*sin(y/71.0),
       0.5+0.5*sin(x*x*x/120000.0+y*y/1700.0)
      );
}
```

Sending vertices into the pipeline: example 2

```
VertexArray *va_cube = NULL;
Buffer *buf_cube_vertices = NULL;
Buffer *buf_cube_faceId = NULL;
......................
GLfloat cube_vertices[] = {
  0,1,0,  0,0,0,  0,0,1,
  0,1,0,  0,0,1,  0,1,1,
  0,0,1,  0,0,0,  1,0,0,
  0,0,1,  1,0,0,  1,0,1,
  0,0,0,  0,1,0,  1,1,0,
  0,0,0,  1,1,0,  1,0,0,
  1,0,1,  1,0,0,  1,1,0,
  1,1,0,  1,1,1,  1,0,1,
  1,1,0,  0,1,0,  0,1,1,
  0,1,1,  1,1,1,  1,1,0,
  1,1,1,  0,1,1,  0,0,1,
  0,0,1,  1,0,1,  1,1,1
};
GLubyte cube_faceId[] = {
  0,0,0,0,0,0,
  1,1,1,1,1,1,
  2,2,2,2,2,2,
  3,3,3,3,3,3,
  4,4,4,4,4,4,
  5,5,5,5,5,5
};
......................
```

```
  buf_cube_vertices = new Buffer(3,36,cube_vertices);
  buf_cube_faceId = new Buffer(1,36,cube_faceId);
  va_cube = new VertexArray;
  va_cube->attachAttribute(0,buf_cube_vertices);
  va_cube->attachAttribute(1,buf_cube_faceId);
......................
  va_cube->sendToPipeline(GL_TRIANGLES,0,36);
```

Example: the cube programs

```
#version 420

layout (location=0) in vec3 modelCoord;
layout (location=1) in uint faceId;

flat out uint face_id;

uniform mat4 MV;
uniform mat4 P;
uniform vec2 T;

void main()
{
  face_id = faceId;
  vec4 worldCoord = MV *
     vec4(0.4*(modelCoord-vec3(0.5,0.5,0.5))
      +vec3(T,0),1.0);
  gl_Position = P * worldCoord;
}
```

```
#version 420

flat in uint face_id;

out vec3 fragcolor;

void main()
{
  switch(face_id)
  {
    case 0: fragcolor = vec3(1,0,0); break;
    case 1: fragcolor = vec3(0,1,0); break;
    case 2: fragcolor = vec3(0,0,1); break;
    case 3: fragcolor = vec3(1,1,0); break;
    case 4: fragcolor = vec3(1,1,1); break;
    case 5: fragcolor = vec3(1,0,1); break;
    default: fragcolor = vec3(.5,.5,.5); break;
  }
}
```

GLM library

- A header-only library that provides useful math functions
  - Vector operations
  - Matrix operations
  - Replicates functionality of most useful obsolete OpenGL functions such as transformation calls
  - Also, replicates most useful deprecated GL utilities (GLU) functions

Using GLM to set uniform matrix variables

- Setting the modelview and projection matrices
  - Suppose in the shader(s) of program `prog` you have:

```
uniform mat4 P;
uniform mat4 MV;
```

  - Example host code:

```
mat4 PMat = perspective(8.0f,1.0f,15.0f,25.0f);
mat4 MVMat = translate(mat4(),vec3(0.0,0.0,-20.0)) *
    rotate(mat4(),angle1,vec3(1.0,2.0,3.0)) *
    rotate(mat4(),angle2,vec3(-2.0,-1.0,0.0)) *
    scale(mat4(),vec3(0.1,0.1,0.1)) *
    translate(mat4(),vec3(-c.x,-c.y,-c.z));
prog->setUniform("P",&Pmat[0][0]);
prog->setUniform("MV",&MVMat[0][0]);
```

Using GLM to set uniform matrix variables

- Setting the normal matrix
  - In many cases (including this project) the normals are changed only by the rotational component – thus, we can use the 3×3 linear part of the modelview matrix as the normal matrix, or just use the rotation matrix only
  - Using a plain rotation matrix could lead to slightly faster implementation of flat or Gouraud shading since rotation matrix is an isometry (and hence maps unit vectors into unit vectors)
  - In general, you would take inverse and transpose of `NMat` used below before sending it to the uniform variable
  - Host code:

    ```
    mat3 NMat(MVMat);
    prog->setUniform("NM",&NMat[0][0]);
    ```

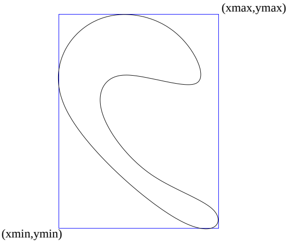
  - Compatible vertex and/or fragment shader uniform declaration:

    ```
    uniform mat3 NMat;
    ```

Setting up the transformations in project 2

- Problem: I am not telling you how big the model is and where it is "centered"
- Solution: scale and move it to a place that the camera can see!
- Modelview matrix and projection matrix are not entirely independent
- First step: "normalize" the model, i.e. scale and translate it so that it fits into a box extending from -1 to 1 in x, y and z

Model Normalization



- Use bounding box
- Compute $x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}$, where $?_{min}$ and $?_{max}$ are the minimum and the maximum ?-coordinates of the model's vertices for $? \in \{x, y, z\}$
- Translate the model by minus center of the bounding box
- Scale by $\frac{2}{\text{maximum dimension of the bounding box}}$

Model Normalization

- Translate the model by minus center of the bounding box,

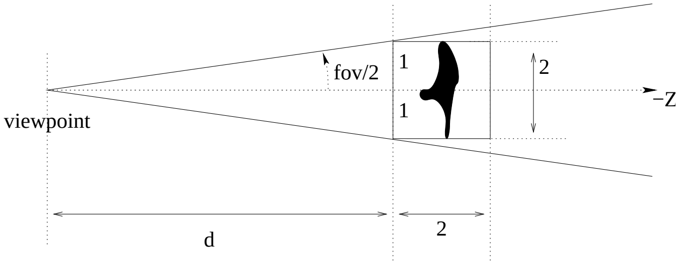
$$\left(-\frac{x_{min}+x_{max}}{2}, -\frac{y_{min}+y_{max}}{2}, -\frac{z_{min}+z_{max}}{2}\right)$$

- Scale by $\frac{2}{\text{maximum dimension of the bounding box}}$, i.e. uniformly by

$$\frac{2}{\max(x_{max}-x_{min}, y_{max}-y_{min}, z_{max}-z_{min})}$$

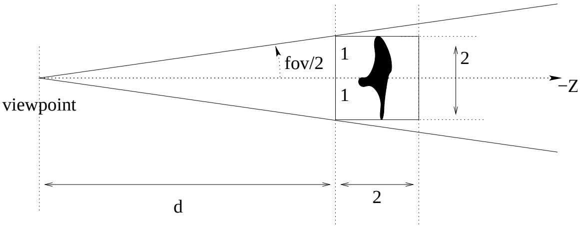
- You may also want to apply rotation matrix (in the project, it will be specified using a trackball interface)
- Make sure order of transformations is correct
- Now it's time for the projection transformation...

Setting up projection



- First, pick your field of view (not too wide, something like 10-20 degrees max)
- In GLM angles (e.g. field of view in `glm::perspective` or rotation angle in `glm::rotate`) are specified in *degrees*; this means you'll likely have to do degree/radian conversion from time to time

Setting up projection



- First, pick your field of view α
- Your model (after normalization) is in the -1...1 box
- From trigonometry, $d = 1/\tan(\alpha/2)$ (see figure)
- Translation by $(0, 0, -1 - d)$ places the -1...1 box in the field of view (good!)
- For distance to front/back clipping planes, one can use $d - 1$ and $d + 3$, to leave some slack for rotation (we don't want to clip the model even if it is rotated around its center)

Setting up transformations: summary

- Pick field of view α
- Modelview transformation:
  - Translate the model by minus center of the bounding box,

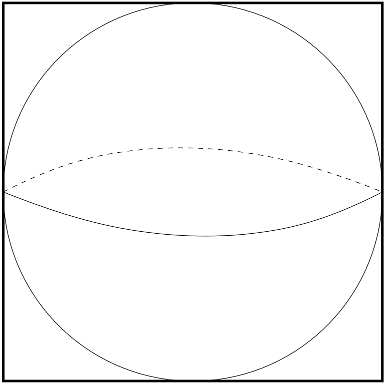
$$\left(-\frac{x_{min}+x_{max}}{2}, -\frac{y_{min}+y_{max}}{2}, -\frac{z_{min}+z_{max}}{2}\right)$$

  - Scale by $\frac{2}{\text{maximum dimension of the bounding box}}$, i.e. uniformly by

$$\frac{2}{\max(x_{max}-x_{min}, y_{max}-y_{min}, z_{max}-z_{min})}$$

  - You may also want to apply rotation matrix (in the project, it will be specified using a trackball interface)
  - Translate by $(0, 0, -1-1/\tan(\alpha/2))$
- Arguments for `glm::project`: field of view α, aspect ratio 1 (square window), front and back clipping planes: $1/\tan(\alpha/2)-1$, $1/\tan(\alpha/2)+3$

Project: trackball interface

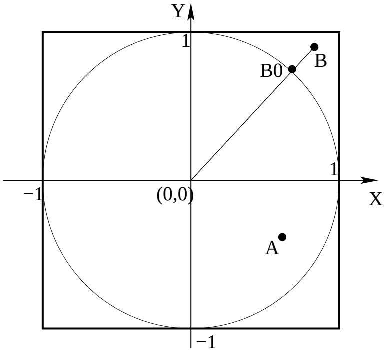


- Imaginary sphere, projecting to circle inscribed into the window
- We'll pretend that parallel projection along z axis is used for the sphere and the sphere is centered at the origin and has unit radius

Project: trackball interface

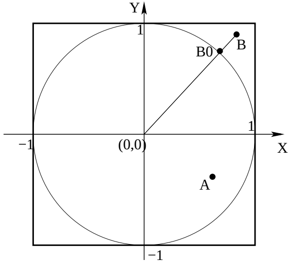
- Say we are rendering into a $d \times d$ window
- Point on the sphere under a pixel (i, j)?
- First, scale i and j to $[-1, 1]$ range (since the sphere is unit...)
  - $x := \frac{2i}{d-1} - 1$, $y := -(\frac{2j}{d-1} - 1)$ (since pixel coordinate $y-$axis points down)
- Now we have x and y coordinates of the point under the pixel (since sphere is projected parallel to z)
- If the viewpoint is at $(0, 0, +\infty)$, then $z = \sqrt{1 - x^2 - y^2}$
- The only complication is that the number under the root may be negative

Project: trackball interface



- This happens if the pixel is outside the projection of the sphere
- If this is the case, first project the pixel to the circle in a radial manner, then use the formula (beware of numerical issues though)

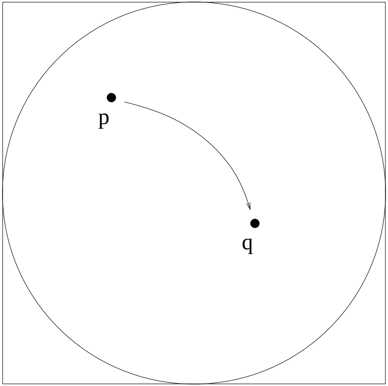
Project: trackball interface



- Radial projection: $B_0 = (x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2})$
- Point under the pixel:

$$\begin{cases} (x, y, \sqrt{1-x^2-y^2}) & \text{if } 1-x^2-y^2 \geq 0 \\ (x/\sqrt{x^2+y^2}, y/\sqrt{x^2+y^2}, 0) & \text{otherwise} \end{cases}$$

Project: trackball interface



- Want rotation that takes p into q (p,q: depend on mouse movement)
- Axis: $\vec{cp} \times \vec{cq}$ where c is the center of the sphere
- Angle: $\angle(\vec{cp}, \vec{cq})$

Trackball interface: breakdown into event handlers

- Event-driven framework provided by the glut library
- Event handlers called not by you, but by glut (from within `glutMainLoop()` function, that basically dispatches events to handlers)
- Of course, you need to tell glut what they are `glutDisplayFunc`, `glutMouseFunc`, `glutMotionFunc`, `glutPassiveMotionFunc` etc.
- Useful global variables
  - R: the superposition of all 'finished' rotations (matrix)
  - R_0: the rotation that is currently being specified using the trackball interface (matrix)
  - i_0, j_0: coordinates of the last mouse button down event
- R and R_0 are initialized to identity at startup.
- Use glm's `mat3` or `mat4` type for R and R_0
- All the mouse event handlers get the coordinates of the mouse cursor (i, j) as arguments.

Trackball interface: breakdown into event handlers

- Mouse button down (begin dragging trackball): assign the event coordinates to i_0 and j_0.
- Mouse moves with button down, current location: (i, j)
  - compute the points p and q on trackball under (i_0, j_0) and (i, j)
  - Set R_0 to the rotation that moves p into q; be sure to use identity as that rotation if $i_0 = i$ and $j_0 = j$
  - Request redraw event by calling `glutPostRedisplay()`; want to give the user immediate feedback
- Mouse button up, current location: (i, j)
  - compute the points p and q on trackball under (i_0, j_0) and (i, j)
  - Let R' be the rotation that moves p into q or identity if $i_0 = i$ and $j_0 = j$
  - $R := R' * R$ (done dragging, so the rotation is finished)
  - $R_0 := Id$ (mouse button up; no rotation being specified)
- Drawing function
  - Apply rotation R and then rotation R_0 when drawing the model

Look at posts in the project directory

- Stray mouse button up events when selecting a menu entry
  - Select menu item by pressing the mouse button [generates menu event]
  - What goes down has to go up (at least a mouse button) [generates mouse button up event]
- Reading the input file
  - Keep it simple
  - Don't worry about the right number of white spaces etc; just read numbers