Geometric Modeling for Computer Graphics

Mike Bailey
mjb@cs.oregonstate.edu

What do we mean by "Modeling"?

How we model geometry depends on what we would like to use the geometry for:

- Looking at its appearance
- Will we need to interact with its shape?
- How does it interact with its environment?
- How does it interact with other objects?
- What is its surface area and volume?
- Will it need to be 3D-printed?
- Etc.

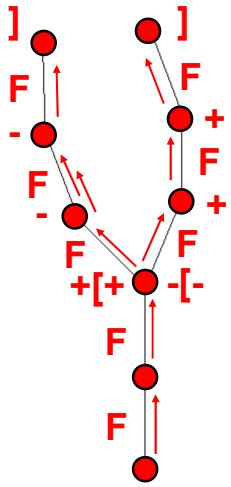
L-Systems as a Special Way to Model 3D Geometry

3

Introduced and developed in 1968 by Aristid Lindenmayer, L-systems are a way to apply grammar rules for generating fractal (self-similar) geometric shapes. For example, take the string:

"FF+[+F-F-F]-[-F+F+F]"

F	move forward one step
+	turn right
-	turn left
[	push state
]	pop state



Oregon State University
Computer Graphics

mjb –August 27, 2024

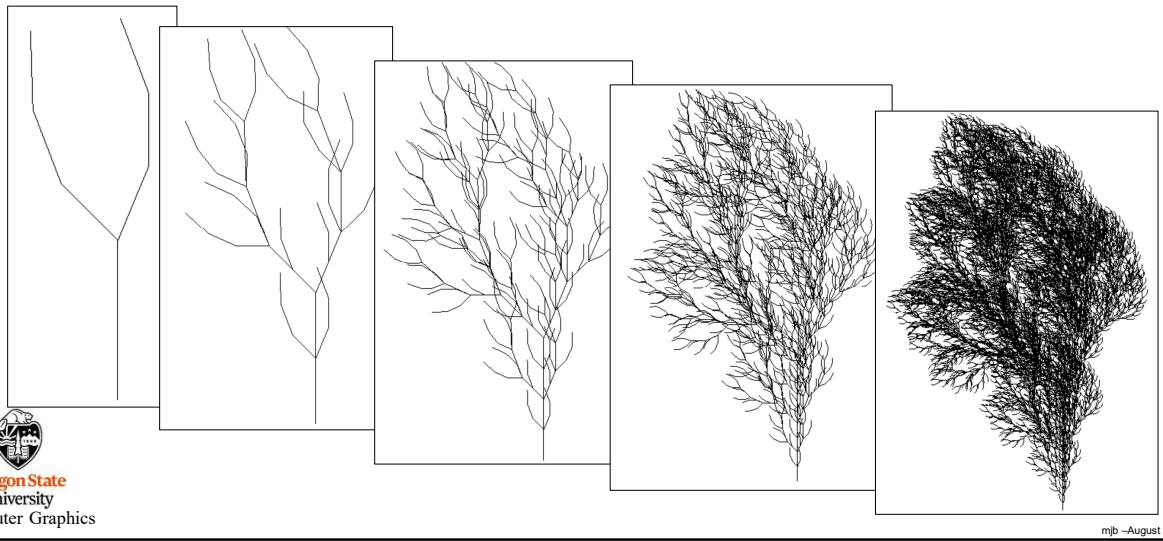
3

L-Systems as a Special Way to Model 3D Geometry

4

But the *real* fun comes when you call that string recursively. For every **F**, replicate that string but with smaller geometry:

"F → FF+[+F-F-F]-[-F+F+F]"



Oregon State University
Computer Graphics

mjb –August 27, 2024

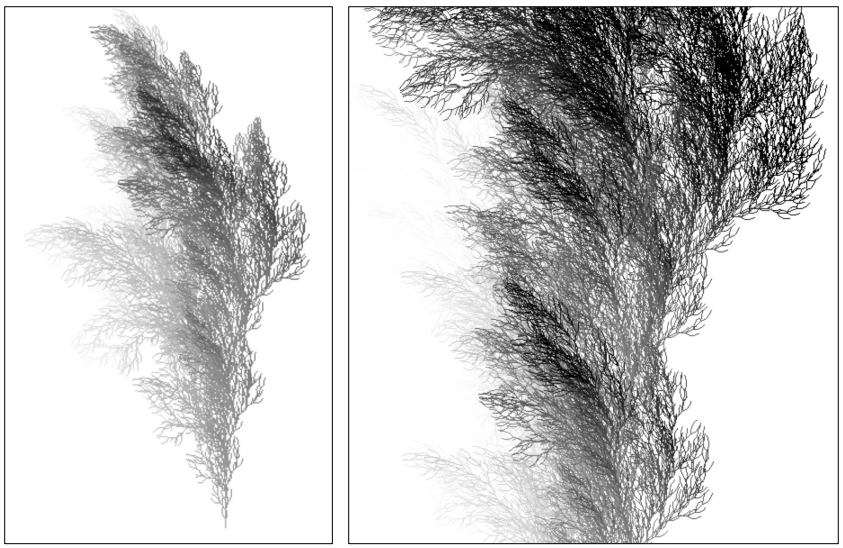
4

L-Systems as a Special Way to Model 3D Geometry

And, of course we can introduce more grammar to swing it into 3D

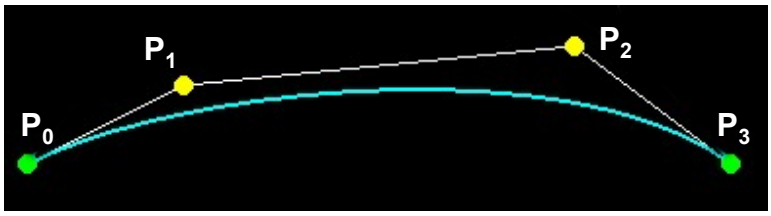
"F → FF+[+F-<F->F]-[-F+^F+vF]"

+	rotate + about Z
-	rotate - about Z
<	rotate + about Y
>	rotate – about Y
v	rotate + about X
^	rotate – about X



Oregon State University
Computer Graphics

Another way to Model:
Curve Sculpting – Bézier Curve Sculpting



$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

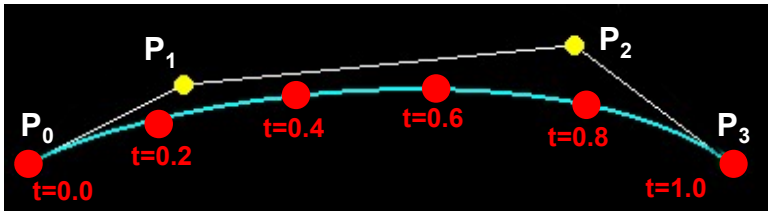
$$0. \le t \le 1.$$

where ***P*** represents $\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$

Oregon State University
Computer Graphics

t goes from 0.0 to 1.0 in whatever increment you'd like

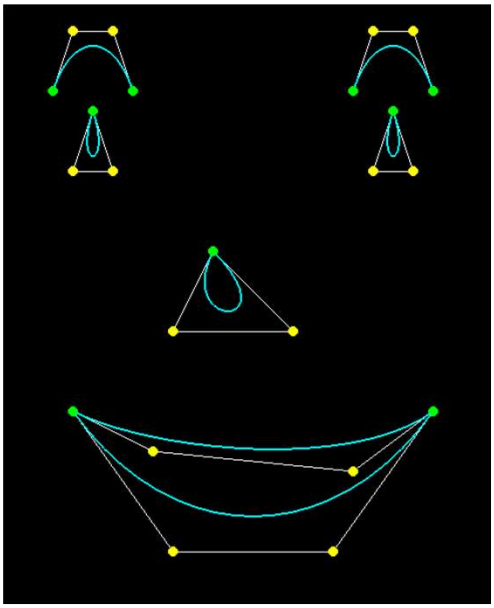
$$0. \le t \le 1.$$

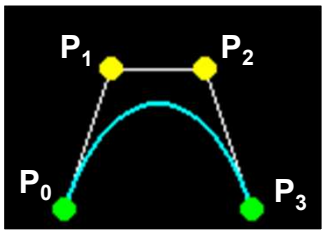


You draw the curve as a series of lines

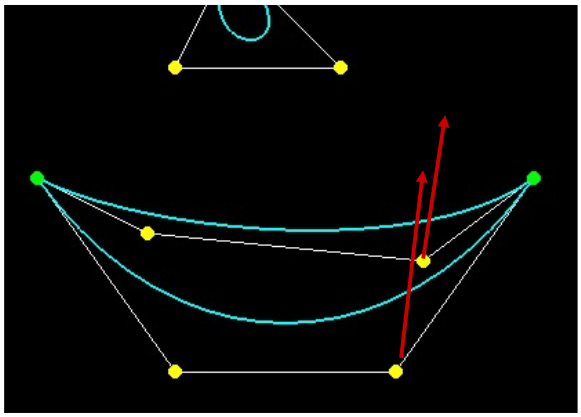
GL_LINE_STRIP is a good topology for this

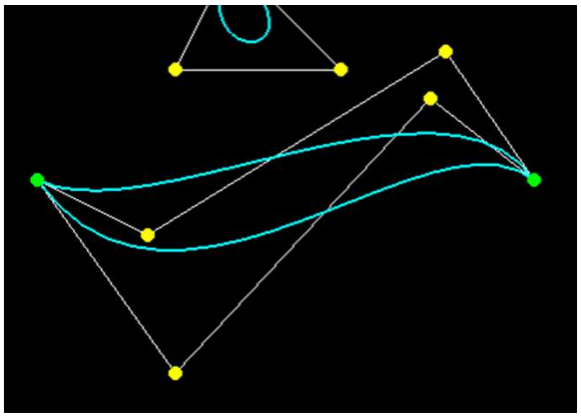
Curve Sculpting – Bézier Curve Sculpting Example





Curve Sculpting – Bézier Curve Sculpting Example

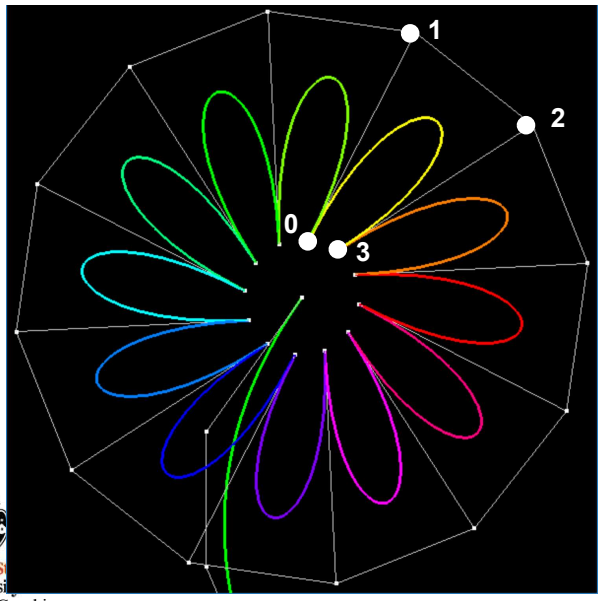


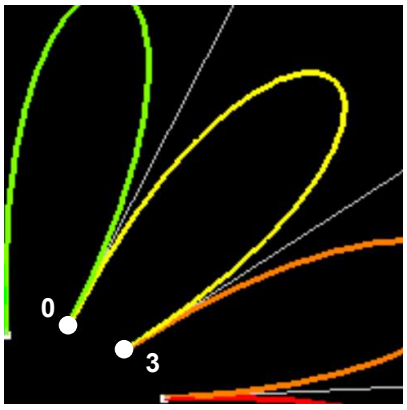


Moving a *single* control point moves its *entire* curve

A *Small* Amount of Input Change Results in a *Large* Amount of Output Change

The Yellow 4-Point Bézier Curve





Another way to Model:
Curve Sculpting – Catmull-Rom Curve Sculpting

The Catmull-Rom curve consists of any number of points.
The first point influences how the curve starts.
The last point influences how the curve ends.
The overall curve goes smoothly through all other points.

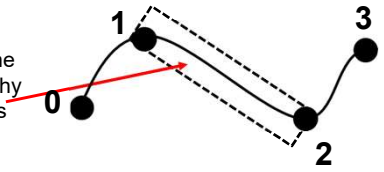
To draw the curve, grab points 0, 1, 2, and 3, call them P_0, P_1, P_2, and P_3, and loop through the following equation, varying t from 0. to 1. in an increment of your own choosing:

$$P(t) = 0.5 * [2 * P_1 + t * (-P_0 + P_2) + t^2(2 * P_0 - 5.* P_1 + 4P_2 - P_3) + t^3(-P_0 + 3P_1 - 3P_2 + P_3)]$$

$$0. \le t \le 1.$$

where ***P*** represents $\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$

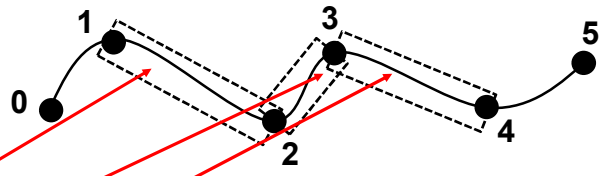
For each set of 4 points, this equation just draws the line between the second and third points. That's why you keep having to use subsequent sets of 4 points



Oregon State University
Computer Graphics

Another way to Model:
Curve Sculpting – Catmull-Rom Curve Sculpting

For each set of 4 points, this equation just draws the line between the second and third points. That's why you keep having to use subsequent sets of 4 points



To draw the curve, grab points 0, 1, 2, and 3, call them P_0, P_1, P_2, and P_3, and loop through the equation, varying t from 0. to 1. in an increment of your own choosing.

Then, grab points 1, 2, 3, and 4, call them P_0, P_1, P_2, and P_3, and loop through the same equation.

Then, grab points 2, 3, 4, and 5, call them P_0, P_1, P_2, and P_3, and loop through the same equation

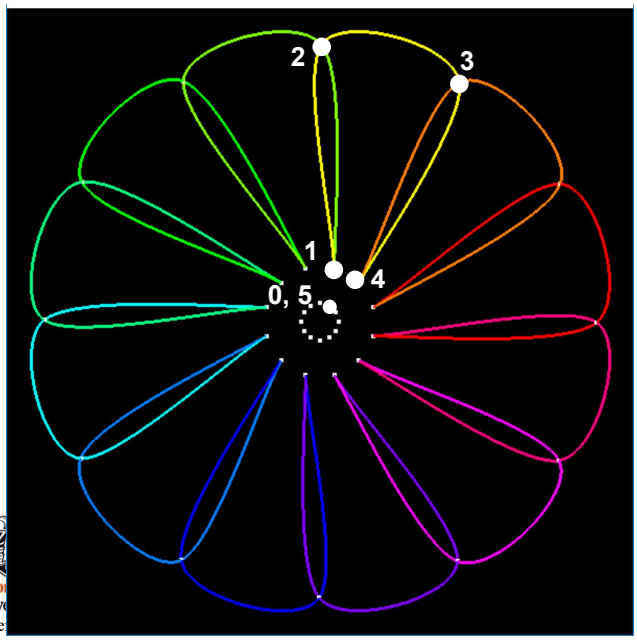
And so on…

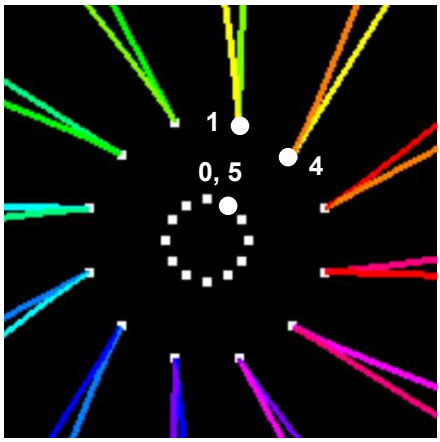
A *Small* Amount of Input Change Results in a *Large* Amount of Output Change

Oregon State University
Computer Graphics

The Yellow 6-Point Catmull-Rom Curve

13





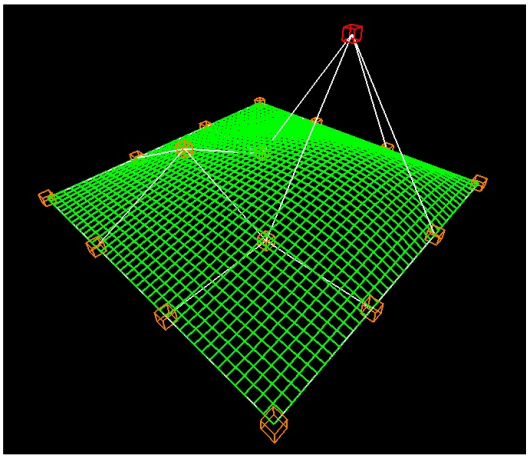
Oregon State University Computer Graphics

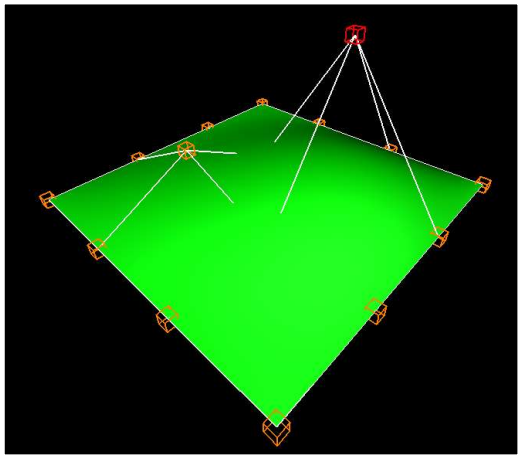
mjb –August 27, 2024

13

Another way to Model: Bézier Surface Sculpting

14





Wireframe **Surface**

Moving a single point moves its entire surface

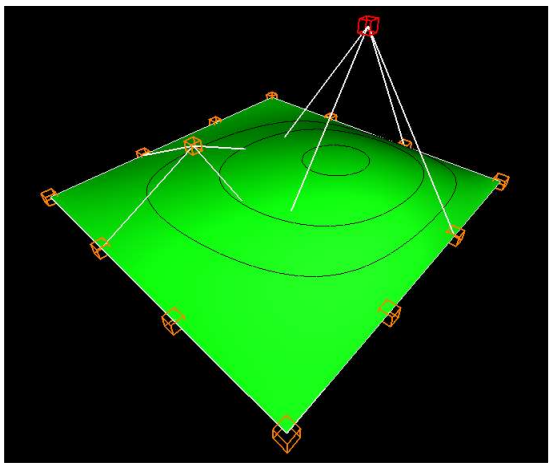
Oregon State University Computer Graphics

A *Small* Amount of Input Change Results in a *Large* Amount of Output Change

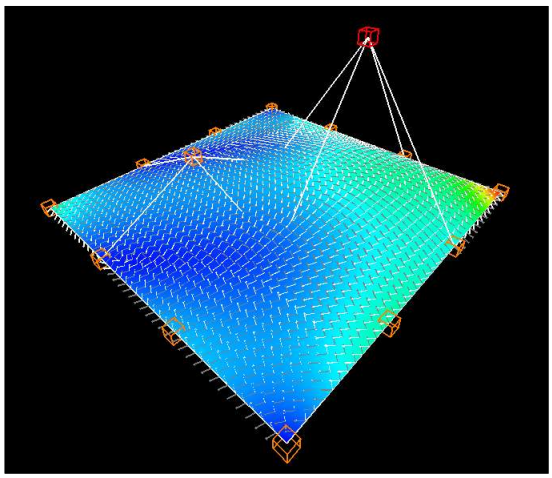
mjb –August 27, 2024

14

Surface Equations can also be used for Analysis



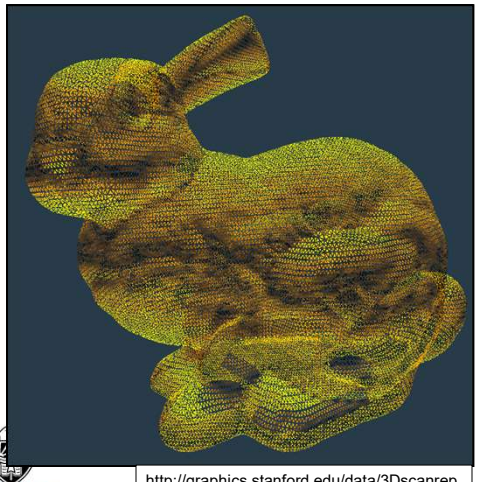
Showing Contour Lines



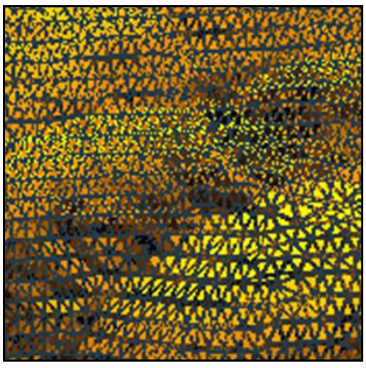
Showing Curvature

Explicitly Listing Geometry and Topology

Models can consist of thousands of vertices and faces – we need some way to list them efficiently



http://graphics.stanford.edu/data/3Dscanrep

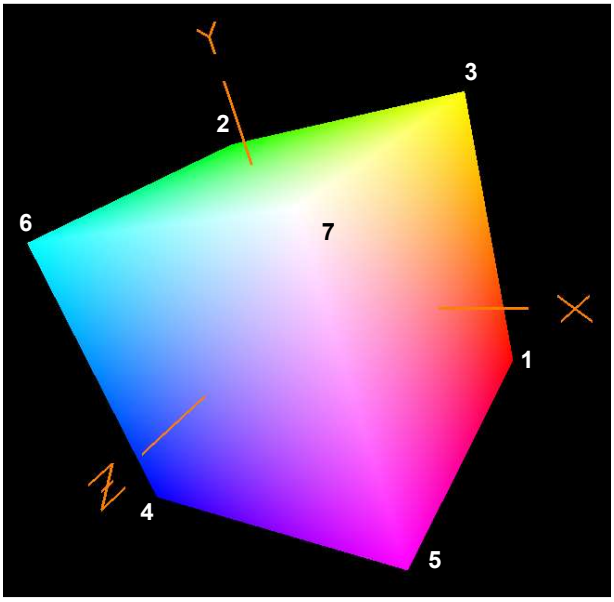


This is called a **Mesh.**

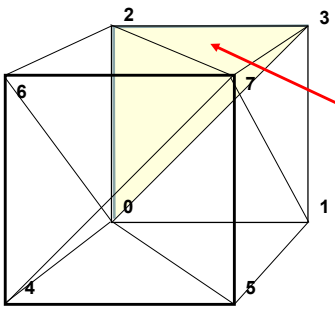
If it's in nice neat rows like this, it is called a **Regular Mesh**.

If it's not, it is called an **Irregular Mesh**, or oftentimes called a **Triangular Irregular Network**, or **TIN**.

Cube Example

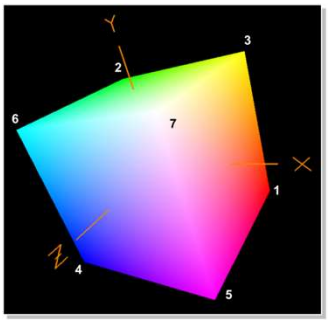


Explicitly Listing Geometry and Topology




```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

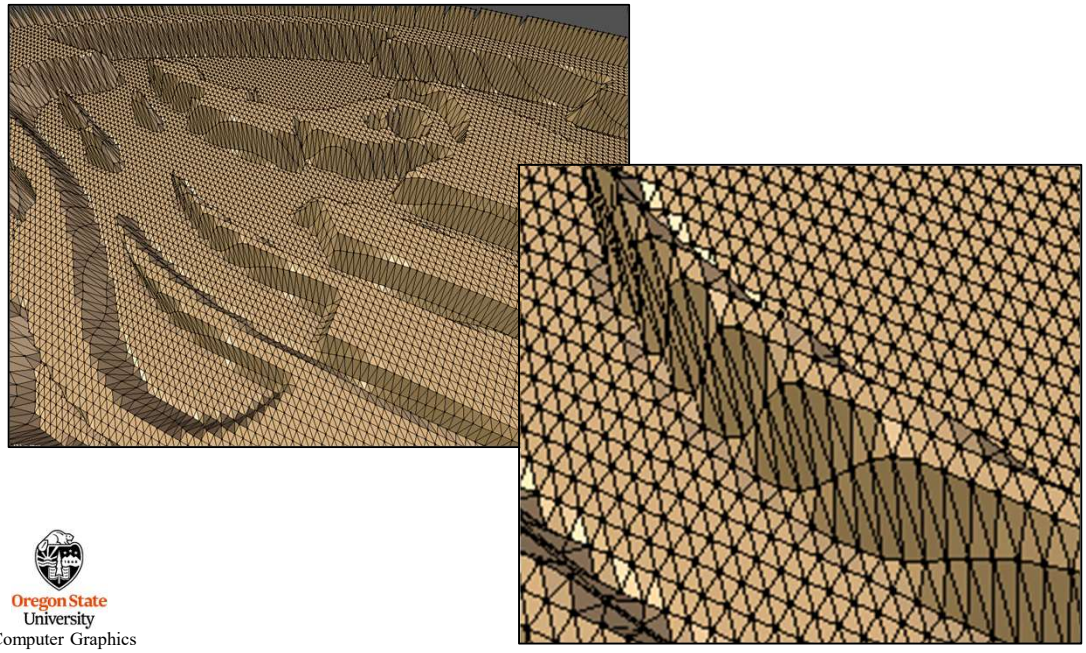
```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```




```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

3D Printing uses an Irregular Triangular Mesh Data Format

19



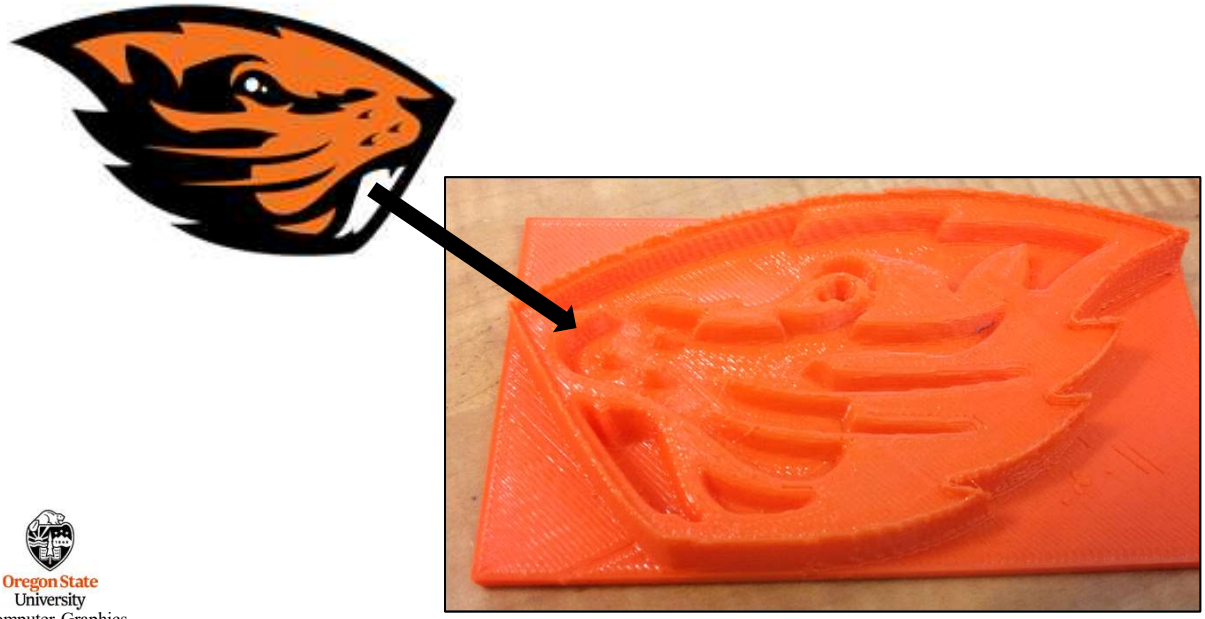
Oregon State University
Computer Graphics

mjb –August 27, 2024

19

3D Printing uses an Irregular Triangular Mesh Data Format

20



Oregon State University
Computer Graphics

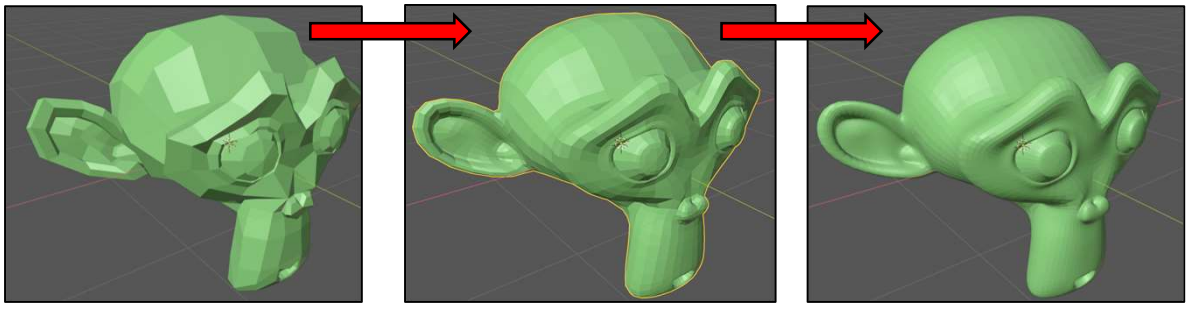
mjb –August 27, 2024

20

Go Beavs – mmmmmm! ☺

Oregon State University
Computer Graphics

Meshes Can Be Smoothed

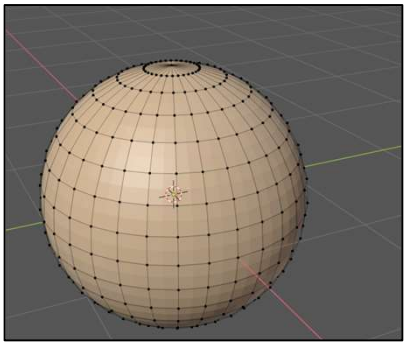


Oregon State University
Computer Graphics

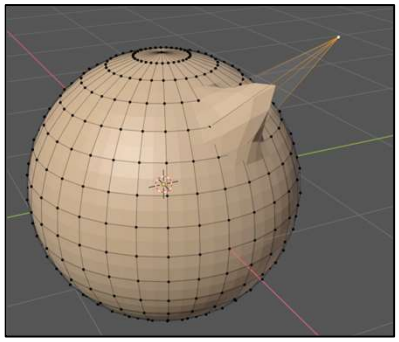
Meshes Can Be Edited

23

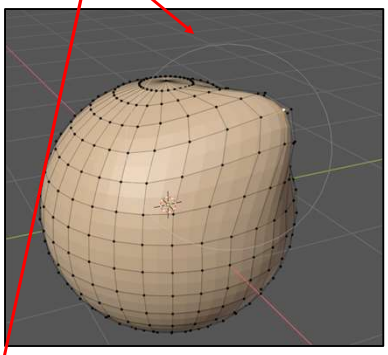
"Circle of Influence"



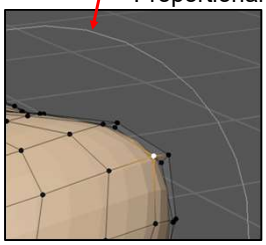
Original



Pulling on a single Vertex



Pulling on a Vertex with Proportional Editing Turned On



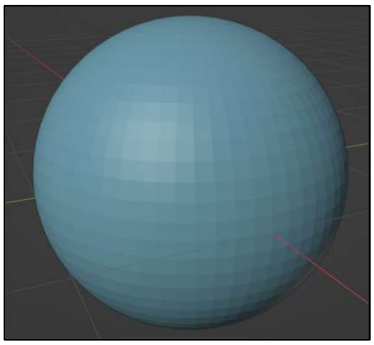
Oregon State University
Computer Graphics

mjb –August 27, 2024

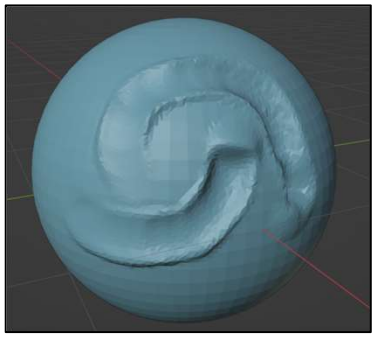
23

Meshes Can Be Sculpted

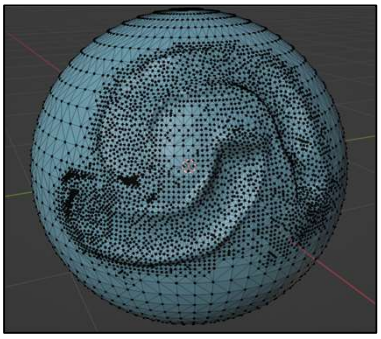
24



Original



"Clay Thumb" Sculpting



Sculpting Can Produce Additional Mesh Vertices

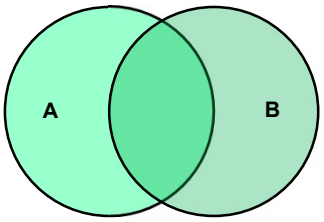
Oregon State University
Computer Graphics

–August 27, 2024

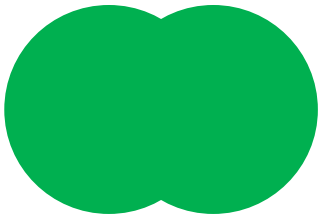
24

25

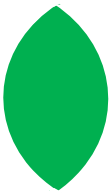
Remember Venn Diagrams (2D Boolean Operators) from High School?



Two Overlapping Shapes

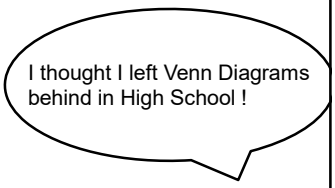


Union: A∪B



Intersection: A∩B

Difference: A-B



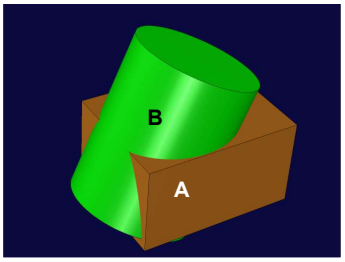
Oregon State University
Computer Graphics

mjb –August 27, 2024

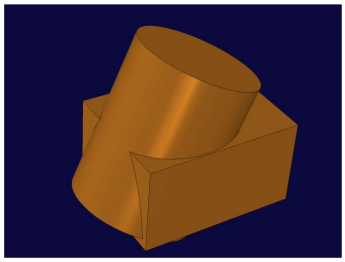
25

26

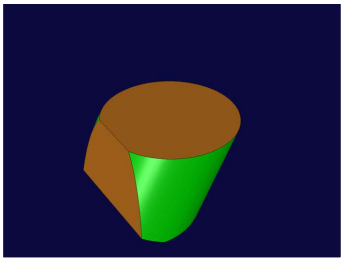
Well, Welcome to Venn Diagrams in 3D



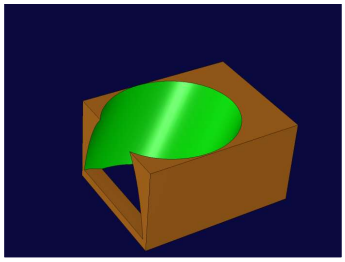
Two Overlapping Solids



Union: A∪B



Intersection: A∩B



Difference: A-B

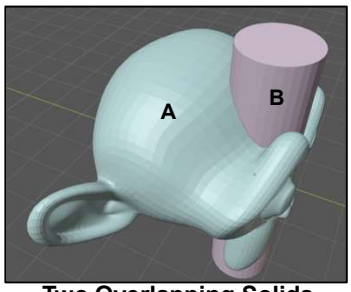
This is often called **Constructive Solid Geometry**, or **CSG**

Oregon State University
Computer Graphics

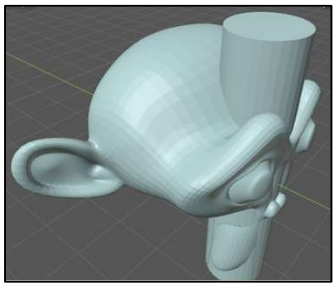
mjb –August 27, 2024

26

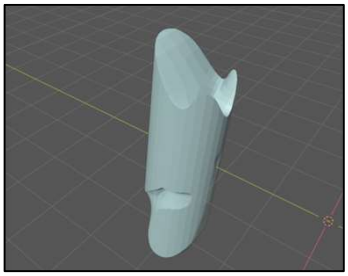
Geometric Modeling Using 3D Boolean Operators on Meshes



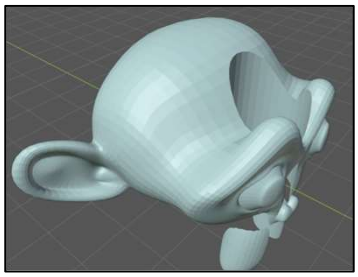
Two Overlapping Solids



Union: A∪B

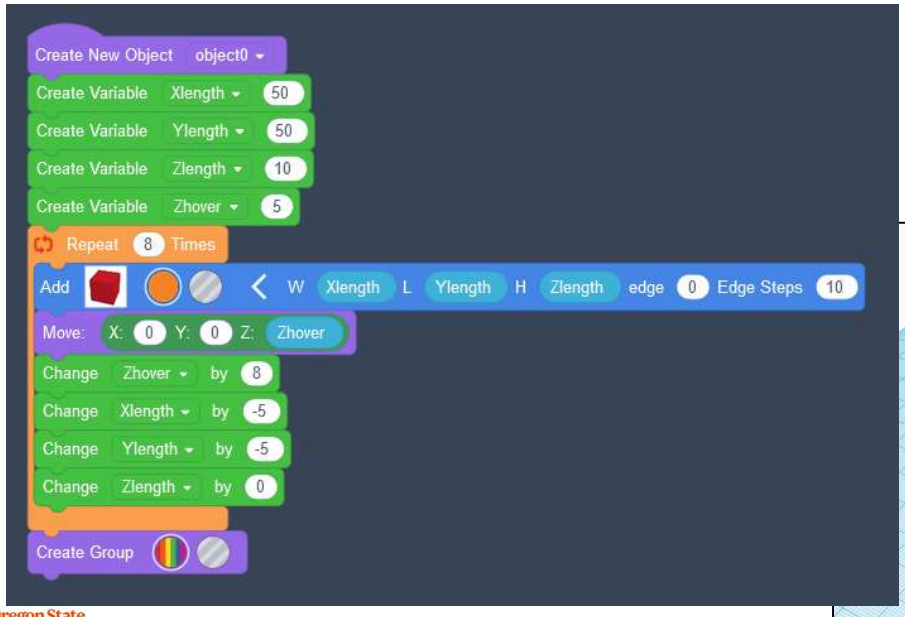


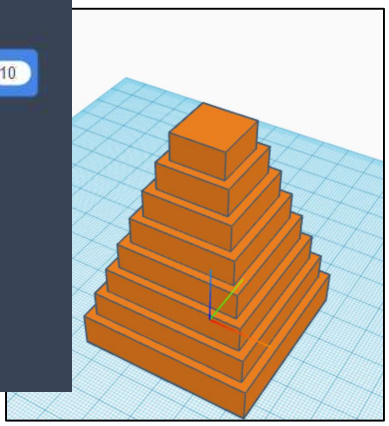
Intersection: A∩B



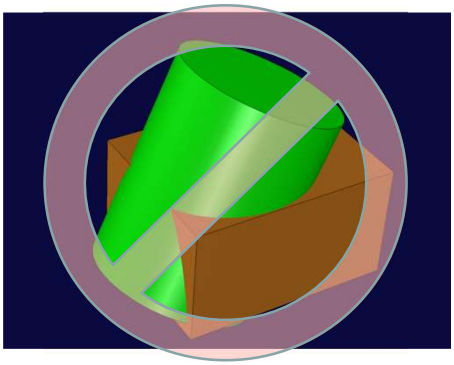
Difference: A-B

Procedural Geometric Modeling Using TinkerCad/Codeblocks

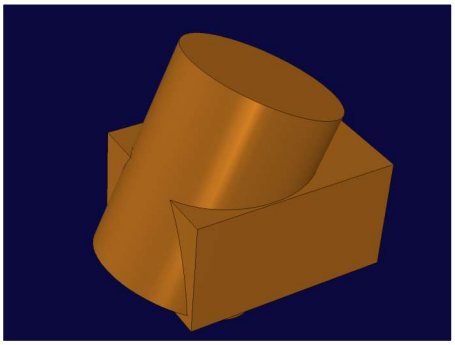




3D Boolean Operators are Important in 3D Printing as well as General Modeling



Two Overlapping Solids – Cannot Be 3D Printed

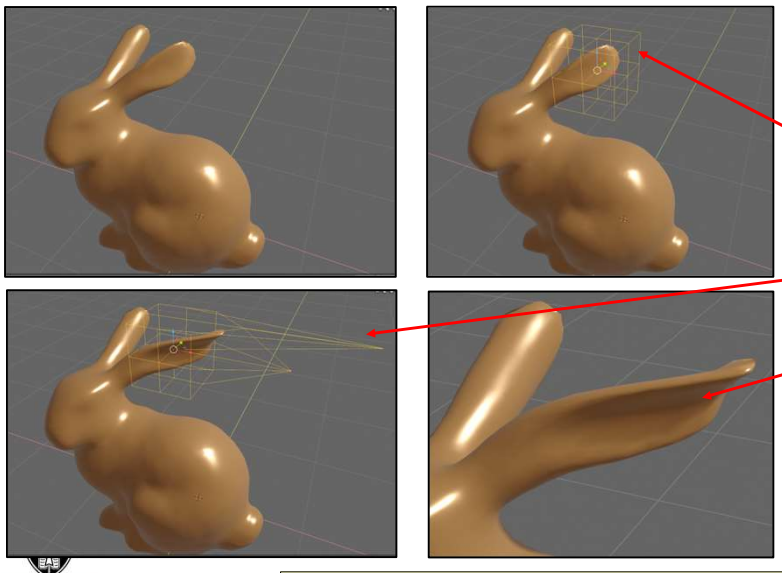


Two Overlapping Solids Unioned – Now Can Be 3D Printed

Oregon State University
Computer Graphics

Intersected and Differenced Solids Can be 3D Printed as Well

Another Way to Edit Meshes: Lattice Sculpting



This is often called a **"Lattice"** or a **"Cage".**

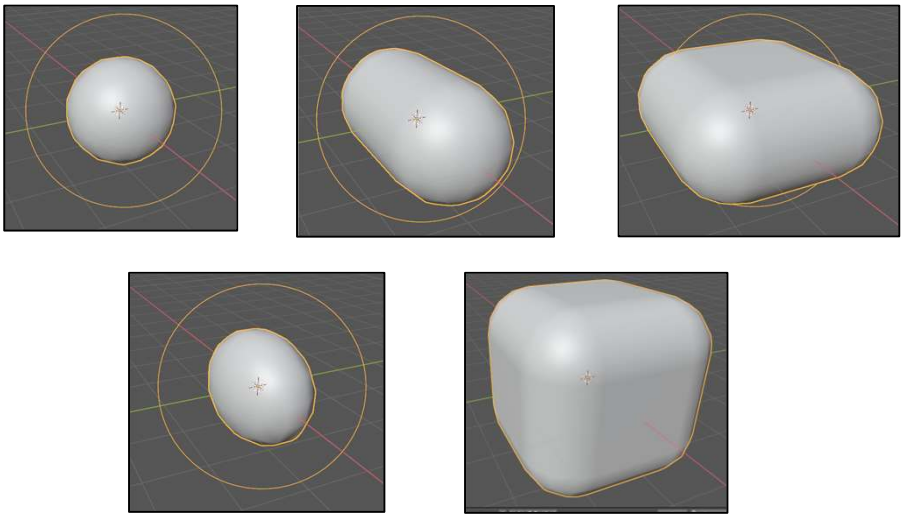
Slip a simpler object (e.g., a subdivided cube) around some of the object's vertices. As you sculpt the simpler object, all those object vertices get sculpted too.

lattice.mp4

Oregon State University
Computer Graphics

A *Small* Amount of Input Change Results in a *Large* Amount of Output Change

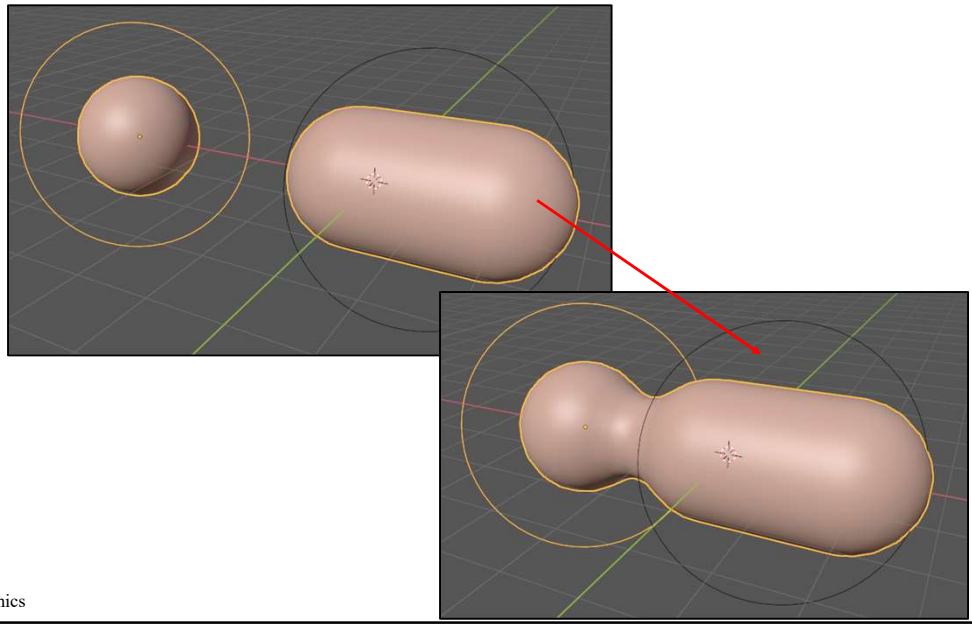
Another Way to Model: Metaball Objects



Oregon State University
Computer Graphics

Metaball Objects

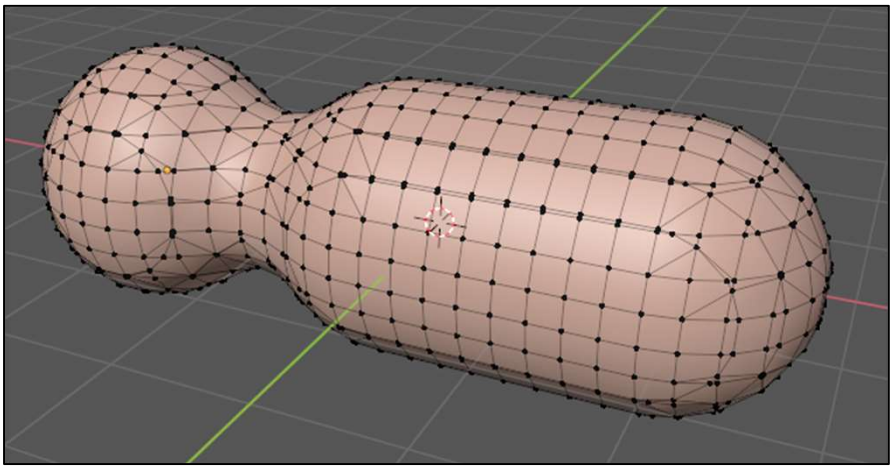
The cool thing is that, if you move them close enough together, they will “glom” into a single object



Oregon State University
Computer Graphics

Metaball Objects Can Be Turned into Meshes for Later Editing

33



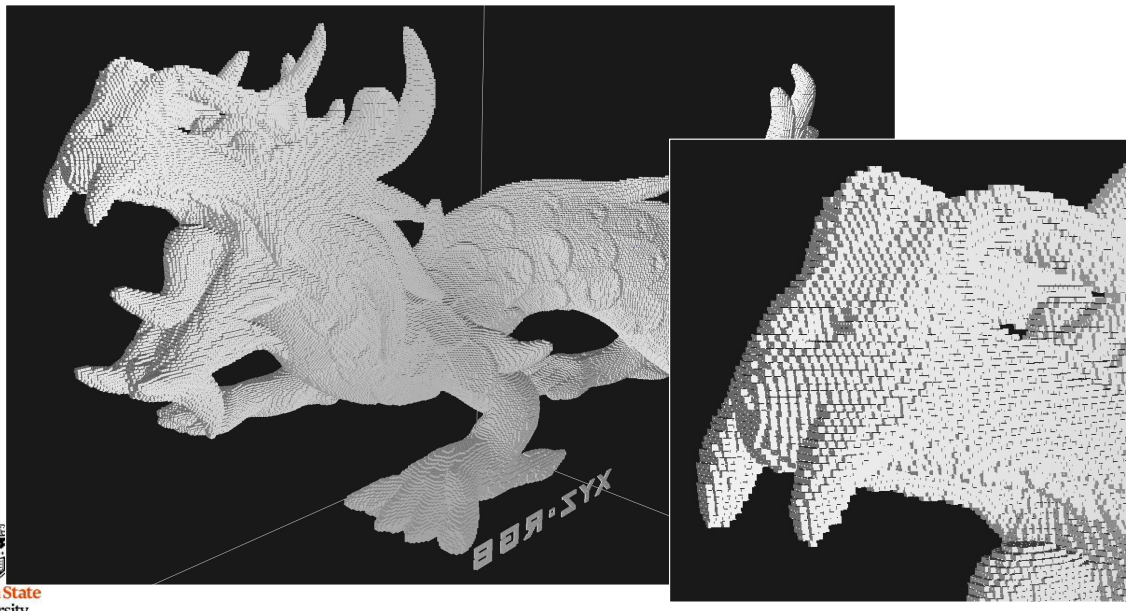
Oregon State University
Computer Graphics

mjb –August 27, 2024

33

Voxelization as a Special Way to Model 3D Geometry

34



Randy Rauwendaal

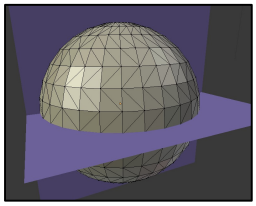
Oregon State University
Computer Graphics

mjb –August 27, 2024

34

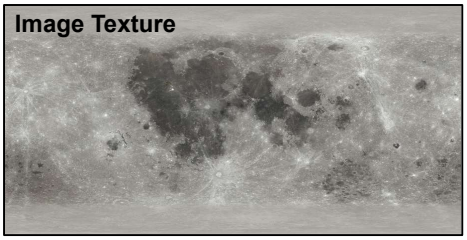
Displacement Textures as a Special Way to Model 3D Geometry

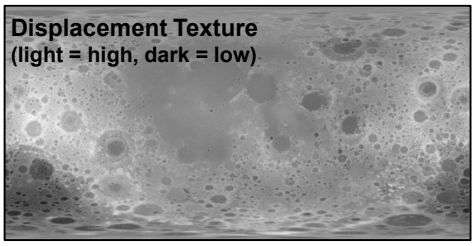
35



Vertex-described Object

+





=

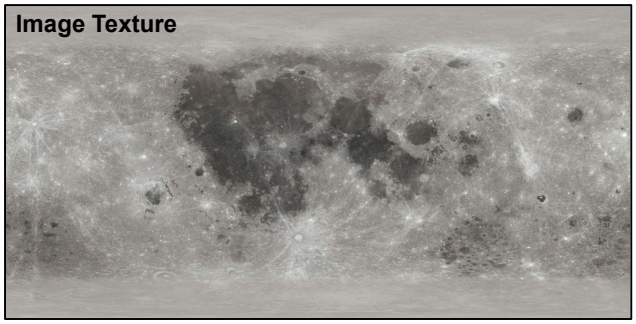
Oregon State University
Computer Graphics

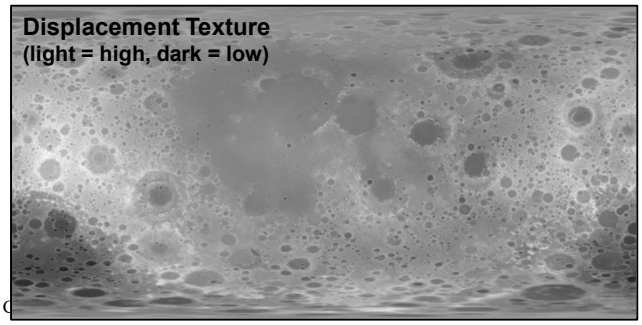
mjb –August 27, 2024

35

Displacement Textures as a Special Way to Model 3D Geometry

36





mjb –August 27, 2024

36

Displacement Textures as a Special Way to Model 3D Geometry

moondisp.vert

```
#version 330 compatibility
uniform float  uLightX, uLightY, uLightZ;
uniform float  uHeightScale;
uniform float  uSeaLevel;
uniform sampler2D          uDispUnit;
uniform bool  uDoElevations;

out vec2       vST;
out vec3       vN;             // normal vector
out vec3       vL;             // vector from point to light

void  main( )
{
        vec2 st = gl_MultiTexCoord0.st;
        vST = st;

        vec3 norm = normalize( gl_NormalMatrix * gl_Normal );          // normal vector
        vN = norm;
        vec3 LightPos = normalize( vec3( uLightX, uLightY, uLightZ ) );
        vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;              // eye coordinate position
        vL = LightPos - ECposition.xyz;                                // vector from the point to the light position

        vec3 vert = gl_Vertex.xyz;
        if( uDoElevations )
        {
                float disp = texture( uDispUnit, st ).r;
                disp -= uSeaLevel;
                disp *= uHeightScale;
                vert += normalize(gl_Normal) * disp;
        }

        gl_Position = gl_ModelViewProjectionMatrix * vec4( vert, 1. );
}
```

Oregon State University
Computer Graphics

Displacement Textures as a Special Way to Model 3D Geometry

moondisp.frag, I

```
#version 330 compatibility
uniform bool               uDoBumpMapping;
uniform float              uKa, uKd;
uniform float              uHeightScale;
uniform float              uNormalScale;
uniform sampler2D          uColorUnit;
uniform sampler2D          uDispUnit;

in vec2                                vST;
in vec3                                vN;
in vec3                                vL;
#define DELTA                         0.01

void main()
{
        vec3 newColor = texture( uColorUnit, vST ).rgb;
        gl_FragColor = vec4( newColor, 1. );
        if( uDoBumpMapping )
        {
                . . .                  // see next slide
        }
}
```

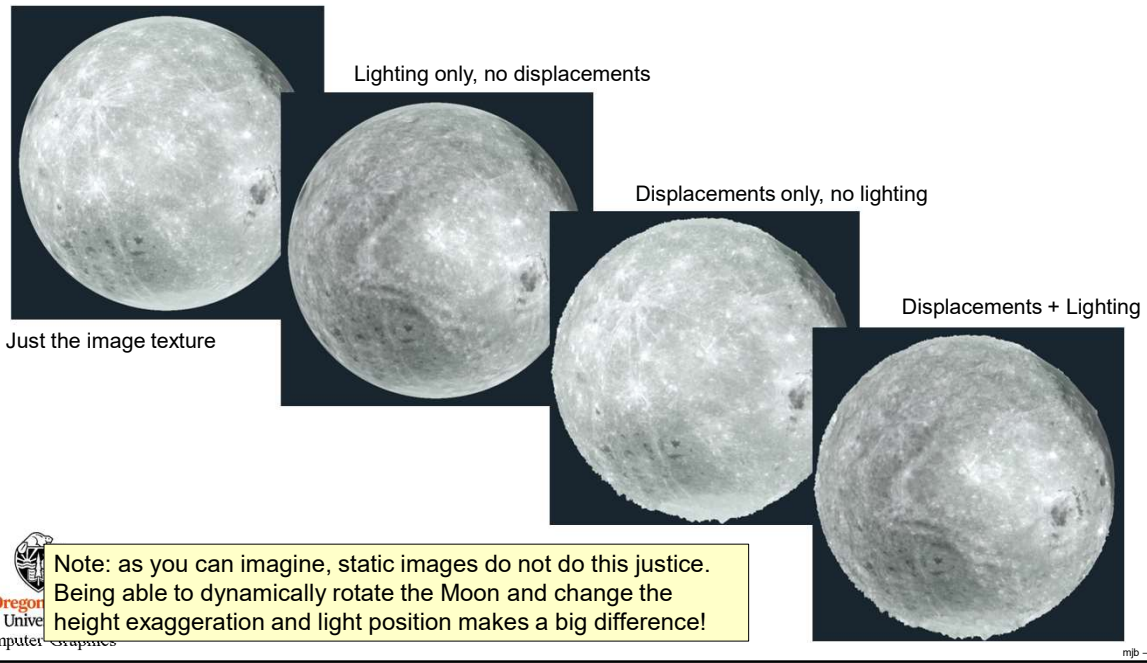
Oregon State University
Computer Graphics

Displacement Textures as a Special Way to Model 3D Geometry

moondisp.frag, II

```
	if( uDoBumpMapping )
	{
		vec2 stp0 = vec2( DELTA, 0. );
		vec2 st0p = vec2( 0. , DELTA );
		float west  = texture2D( uDispUnit, vST-stp0 ).r;
		float east  = texture2D( uDispUnit, vST+stp0 ).r;
		float south = texture2D( uDispUnit, vST-st0p ).r;
		float north = texture2D( uDispUnit, vST+st0p ).r;
		vec3 stangent = vec3( 2.*DELTA, 0., uNormalScale * ( east - west ) );
		vec3 ttangent = vec3( 0., 2.*DELTA, uNormalScale * ( north - south ) );
		vec3 Normal = normalize( cross( stangent, ttangent ) );
		vec3 Light  = normalize(vL);

		vec3 ambient = uKa * newColor;
		float d = 0.;
		if( dot(Normal,Light) > 0. ) // only do diffuse if the light can see the point
		{
			d = dot(Normal,Light);
		}
		vec3 diffuse = uKd * d * newColor;
		gl_FragColor = vec4( ambient+diffuse, 1. );
	}
}
```

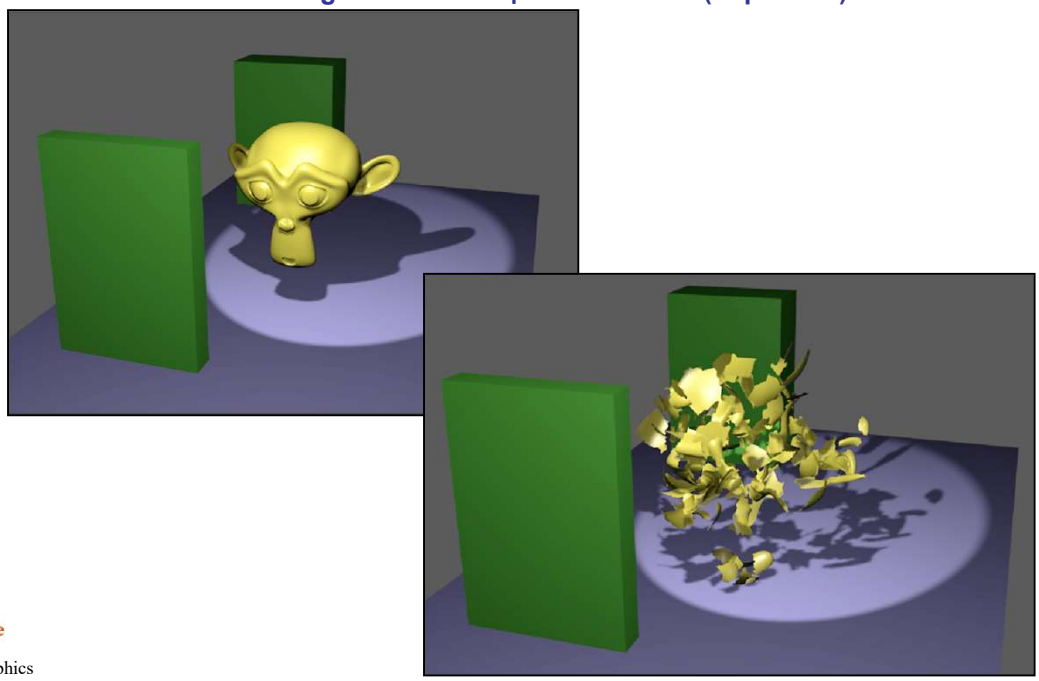

Displacement Textures as a Special Way to Model 3D Geometry



Note: as you can imagine, static images do not do this justice. Being able to dynamically rotate the Moon and change the height exaggeration and light position makes a big difference!

Modeling as an Initial Step in Simulation (Explosion)

41



Oregon State University
Computer Graphics

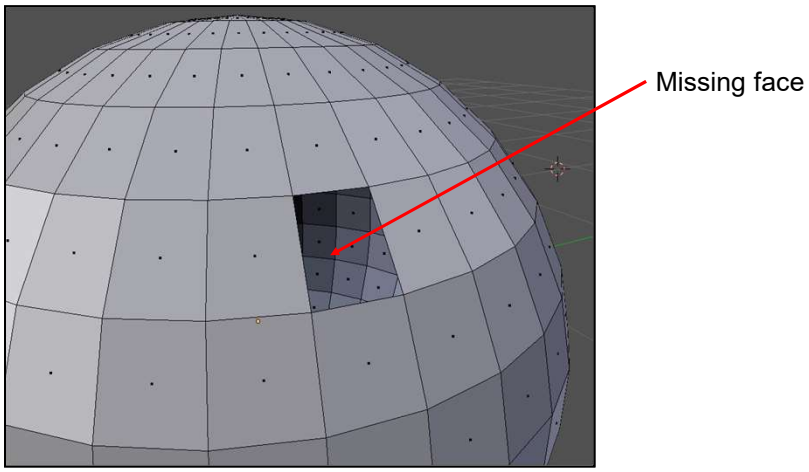
mjb –August 27, 2024

41

Object Modeling Rules for 3D Printing

42

The object must be a legal solid. It must have a definite inside and a definite outside. It can't have any missing face pieces.



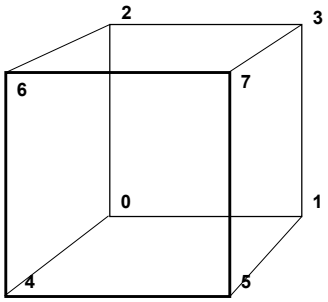
Oregon State University
Computer Graphics

"Definite inside and outside" is sometimes called **"Two-manifold"** or **"Watertight"**

mjb –August 27, 2024

42

The Simplified Euler's Formula* for Legal Solids



*sometimes called the Euler-Poincaré formula

F – E + V = 2

F Faces
E Edges
V Vertices

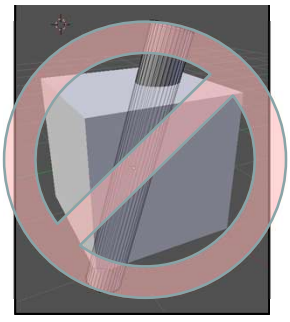
For a cube, 6 – 12 + 8 = 2

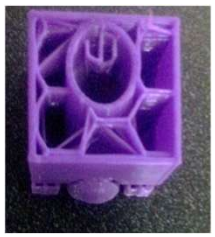
We will talk more about this in the 3D Printing notes!

Object Modeling Rules for 3D Printing

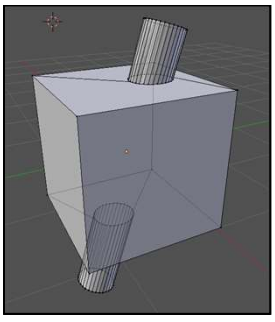
Objects cannot pass through other objects. If you want two shapes together, do a Boolean union on them so that they become one complete object.

Overlapped in 3D -- bad





Boolean union -- good



The OpenGL Mathematics (GLM) Library

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

What is GLM?

GLM is a set of C++ classes and functions that fill in the programming gaps in writing the basic vector and matrix mathematics for computer graphics applications.

GLM isn't a *library* – it is all specified in ***.hpp** header files that get compiled in with your source code.

You can find GLM at:

http://glm.g-truc.net/0.9.8.5/

or you can get a zip file of it on our Class Resources page.

You typically use GLM by putting these lines at the top of your program:

```
#define    GLM_FORCE_RADIANS
#include "glm/vec2.hpp"
#include "glm/vec3.hpp"
#include "glm/mat4x4.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/matrix_inverse.hpp"
#include "glm/gtc/type_ptr.hpp"
```

Why are we even talking about this?

The OpenGL overlords have "deprecated" some of the OpenGL functions we have been using to perform transformations. In the desktop world, it means that the use of such functions is ***discouraged***. In Vulkan and in the mobile world of OpenGL-ES, it means those functions are ***gone***. You might as well become familiar with how to live without them. So, instead of saying:

```
gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );
glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glRotatef( (GLfloat)Xrot, 1., 0., 0. );
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

for OpenGL, you would now say:

```
glm::mat4 modelview;
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3   up(0.,1.,0.);
modelview = glm::lookAt( eye, look, up );
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) );
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) );
modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) );
glMultMatrixf(   glm::value_ptr( modelview )   );
```

Oregon State University
Computer Graphics

Exactly the same concept, but a different expression of it. Read on for details …

The Most Useful GLM Variables, Operations, and Functions

// constructor:

```
glm::mat4( 1. );                    // identity matrix
glm::vec4( );
glm::vec3( );
```

GLM recommends that you use the **"glm::"** syntax and not use **"using namespace"** syntax because they have not made any effort to create unique function names

// multiplications – the * operator has been overloaded:

```
glm::mat4  *  glm::mat4
glm::mat4  *  glm::vec4
glm::mat4  *  glm::vec4(  glm::vec3, 1.  )        // promote vec3 to a vec4 via a constructor
```

// emulating OpenGL transformations *with concatenation*:

```
glm::mat4 glm::rotate(  glm::mat4 const & m,  float angle,  glm::vec3 const & axis );

glm::mat4 glm::scale(  glm::mat4 const & m,  glm::vec3 const & factors );

glm::mat4 glm::translate(  glm::mat4 const & m,  glm::vec3 const & translation );
```

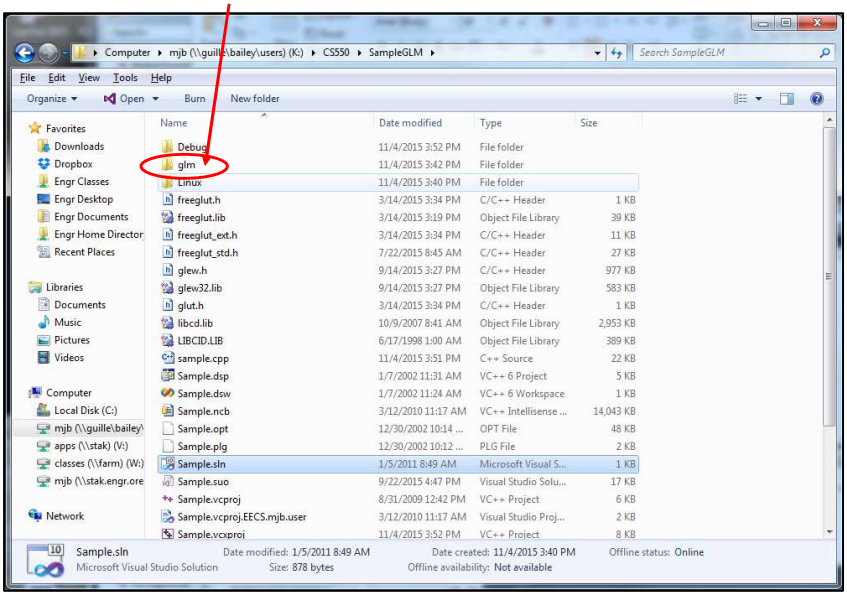
Oregon State University
Computer Graphics

The Most Useful GLM Variables, Operations, and Functions

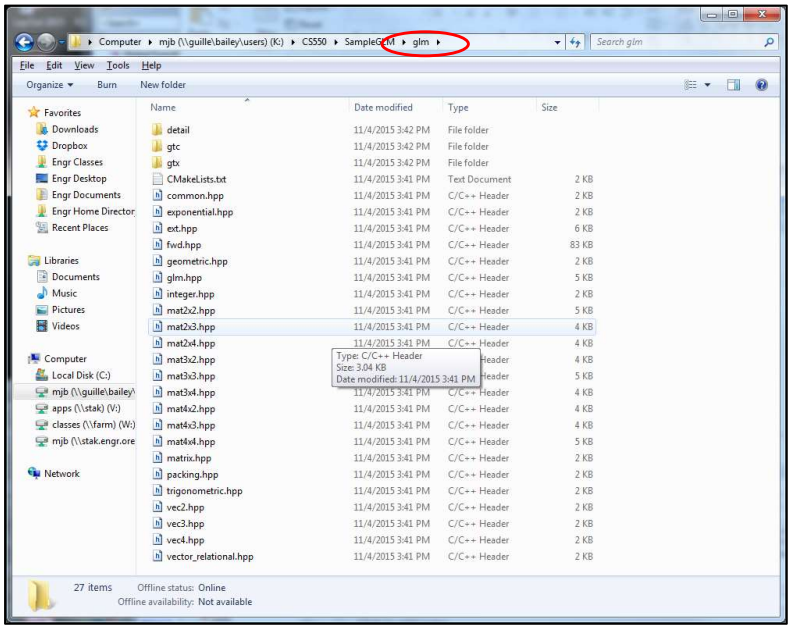

```
// viewing volume  (assign, not concatenate):

glm::mat4  glm::ortho(  float left,  float right,  float bottom,  float top,  float near,  float far );
glm::mat4  glm::ortho(  float left,  float right,  float bottom,  float top );

glm::mat4  glm::frustum(  float left,  float right,  float bottom,  float top,  float near,  float far );
glm::mat4  glm::perspective(  float fovy,  float aspect,  float near,  float far);


// viewing  (assign, not concatenate):

glm::mat4  glm::lookAt(  glm::vec3 const & eye,  glm::vec3 const & look,  glm::vec3 const & up );



// loading matrices into opengl:

glLoadMatrix(   glm::value_ptr(  glm::mat4 )   );

glUniformMatrix4fv(  Location, 1, GL_FALSE,   glm::value_ptr( glm::mat4 )  );
```


Installing GLM into your own space

I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.



Here's what that GLM folder looks like



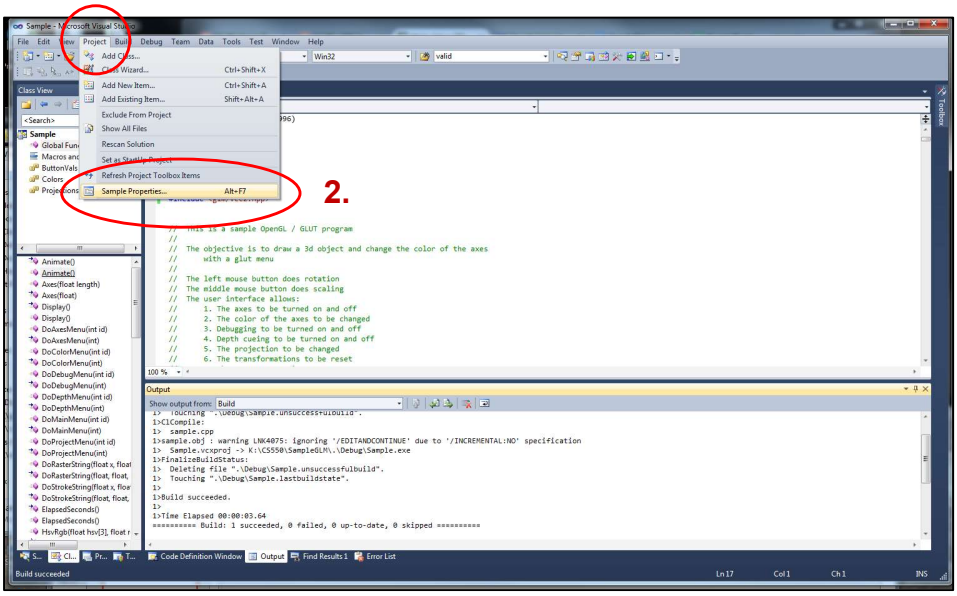
Telling Linux about where the GLM folder is

g++ … **-I.** …

"minus-capital-eye-period" means "also look for the < > includes in this folder"

Instead of the period, you can list a full or relative pathname.

Telling Visual Studio about where the GLM folder is

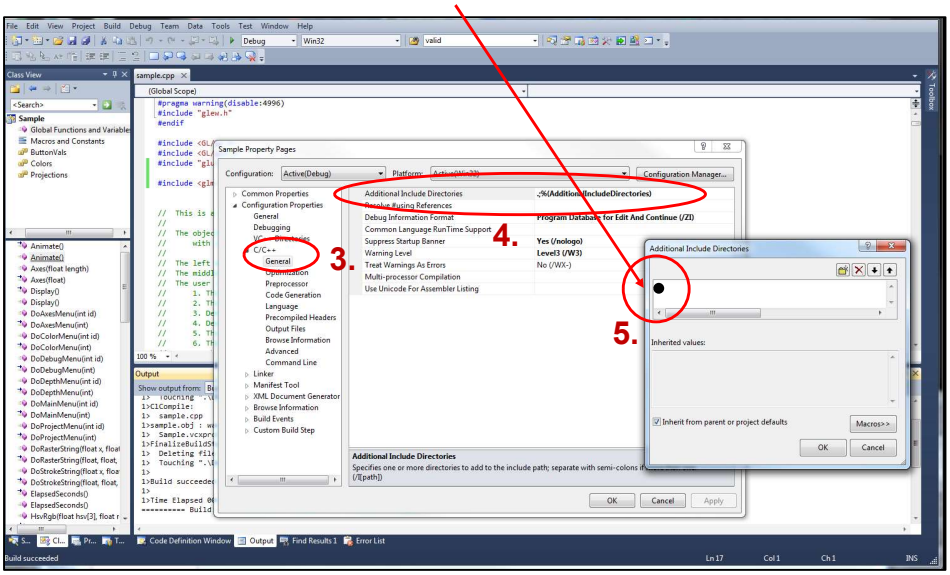


Telling Visual Studio about where the GLM folder is

A ***period***, indicating that the **project folder** should also be searched when a

#include <xxx>

is encountered. If you put it somewhere else, enter that full or relative path instead.



Using Transformations, OpenGL-style, like in the sample.cpp Program


```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
if( WhichProjection == ORTHO )
        glOrtho( -3., 3.,    -3., 3.,    0.1, 1000. );
else
        gluPerspective( 90., 1.,       0.1, 1000. );

// place the objects into the scene:
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

// set the eye position, look-at position, and up-vector:
gluLookAt( 0., 0., 3.,    0., 0., 0.,    0., 1., 0. );

// rotate the scene:
glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glRotatef( (GLfloat)Xrot, 1., 0., 0. );

// uniformly scale the scene:
if( Scale < MINSCALE )
        Scale = MINSCALE;
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

Oregon State University
Computer Graphics

Using Transformations, GLM-style, I

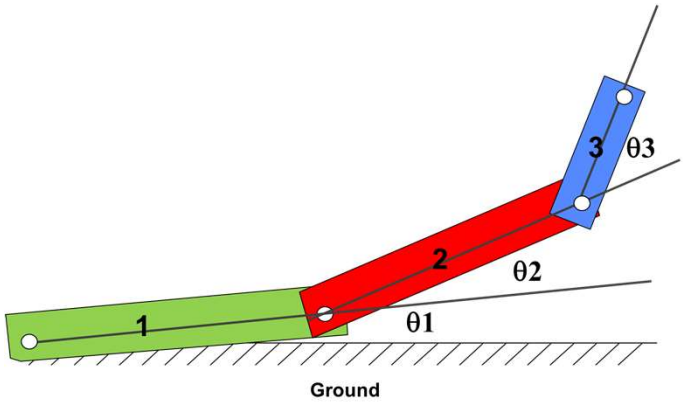

```
#include <glm/vec3.hpp>
#include <glm/mat4x4.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

        . . .

// convert degrees to radians:
const float D2R = M_PI/180.f;                    // 0.01745…

        . . .

glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
glm::mat4  projection;

if( WhichProjection == ORTHO )
        projection = glm::ortho( -3., 3.,    -3., 3.,    0.1, 1000. );
else
        projection = glm::perspective( D2R*90., 1., 0.1, 1000. );

// apply the projection matrix:
glMultMatrixf(   glm::value_ptr( projection )   );
```

Oregon State University
Computer Graphics

Using Transformations, GLM-style, II


```
// place the objects into the scene:
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

// set the eye position, look-at position, and up-vector:
glm::vec3  eye(0.,0.,3.);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
glm::mat4  modelview = glm::lookAt( eye, look, up );

// rotate the scene (warning -- unlike OpenGL's glRotatef,
//        GLM's rotate method takes angles in *radians*):
modelview = glm::rotate( modelview, D2R*Yrot,  glm::vec3(0.,1.,0.) );
modelview = glm::rotate( modelview, D2R*Xrot,  glm::vec3(1.,0.,0.) );

// uniformly scale the scene:
if( Scale < MINSCALE )
        Scale = MINSCALE;
modelview = glm::scale( modelview,  glm::vec3(Scale,Scale,Scale) );

// apply the modelview matrix:
glMultMatrixf(   glm::value_ptr( modelview )   );
```


Let's Re-work the Forward Kinematics Program Using GLM



$$[M_{3/G}] = [T_{1/G}] * [R_{\theta 1}] * [T_{2/1}] * [R_{\theta 2}] * [T_{3/2}] * [R_{\theta 3}]$$

$$[M_{3/G}] = [M_{1/G}] * [M_{2/1}] * [M_{3/2}]$$

Let's Re-work the Forward Kinematics Program Using GLM

Standard OpenGL

```
void DrawMechanism( float θ1, float θ2, float θ3 )
{
    glPushMatrix( );
        glRotatef( θ1,  0., 0., 1. );
        DrawLinkOne( );

        glTranslatef( LENGTH_1, 0., 0. );
        glRotatef( θ2,  0., 0., 1. );
        DrawLinkTwo( );

        glTranslatef( LENGTH_2, 0., 0. );
        glRotatef( θ3,  0., 0., 1. );
        DrawLinkThree( );
        glPopMatrix( );
}
```

Using GLM

```
void DrawMechanism( float θ1, float θ2, float θ3 )
{
    glm::mat4  identity = glm::mat4( 1. );
    glm::vec3  zaxis    = glm::vec3(0.,0.,1.);
    glPushMatrix( );
        glm::mat4 m1g = glm::rotate( identity, θ1,  zaxis );
        glMultMatrixf(  glm::value_ptr( m1g )  );
        DrawLinkOne( );

        glm::mat4  m21  =  glm::translate( m1g, glm::vec3( LENGTH_1, 0., 0. ) );
        m21 = glm::rotate( m21, θ2,  zaxis );
        glMultMatrixf(  glm::value_ptr( m21 )  );
        DrawLinkTwo( );

        glm::mat4  m32  =  glm::translate( m1g, glm::vec3( LENGTH_2, 0., 0. ) );
        m32 = glm::rotate( m32, θ3,  zaxis );
        glMultMatrixf(  glm::value_ptr( m32 )  );
        DrawLinkThree( );
    glPopMatrix( );
}
```

$$[M_{3/G}] = [M_{1/G}] * [M_{2/1}] * [M_{3/2}]$$

Oregon State University
Computer Graphics

Passing GLM Matrices into a Vertex Shader

In the shader:

```
uniform mat4       projectionMatrix;
uniform mat4       viewMatrix;
uniform mat4       modelMatrix;

mat4 PVM = projectionMatrix * viewMatrix * modelMatrix;
gl_Position = PVM * gl_Vertex;
```

In the C/C++ program:

```
glm::mat4  projection = glm::perspective( D2R*90., 1., 0.1, 1000. );

glm::vec3  eye(0.,0.,3.);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
glm::mat4  view  =  glm::lookAt( eye, look, up );

glm::mat4  model( 1. );      // identity
model = glm::rotate( model, D2R*Yrot,  glm::vec3(0.,1.,0.) );
model = glm::rotate( model, D2R*Xrot,  glm::vec3(1.,0.,0.) );

Pattern.Use( );
Pattern.SetUniformVariable( "projectionMatrix", projection );
Pattern.SetUniformVariable( "viewMatrix", view );
Pattern.SetUniformVariable( "modelMatrix", model );
```

Oregon State University
Computer Graphics

GLM for Vulkan

```
glm::mat4  projection = glm::perspective( D2R*90., 1., 0.1, 1000. );
projection[1][1]  *=  -1.;          // Vulkan's projected Y is inverted from OpenGL's

glm::vec3  eye(0.,0.,3.);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);
glm::mat4  view  =  glm::lookAt( eye, look, up );

glm::mat4  model( 1. );  // identity
model = glm::rotate( model, D2R*Yrot,  glm::vec3(0.,1.,0.) );
model = glm::rotate( model, D2R*Xrot,  glm::vec3(1.,0.,0.) );
```

1

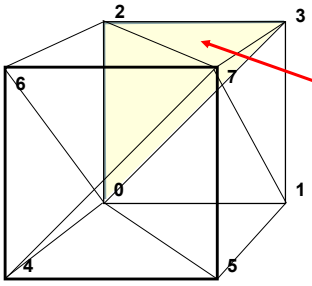
Vertex Buffer Objects

Mike Bailey
mjb@cs.oregonstate.edu

1

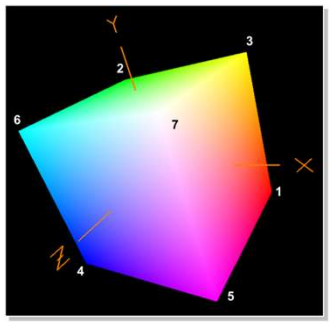
Remember this from the Geometric Modeling Notes?

2




```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```

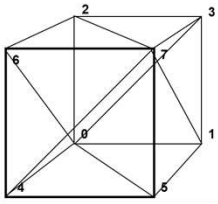



```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```


2

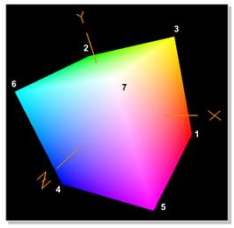
Vertex Buffer Objects: The Big Idea

- Store vertex coordinates and vertex attributes on the ***graphics card***.
- Optionally store the connections on the graphics card too.
- Every time you go to redraw, coordinates will be pulled from GPU memory instead of CPU memory, avoiding a significant amount of bus latency.




```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 },
        { 0, 5, 4 }
};
```

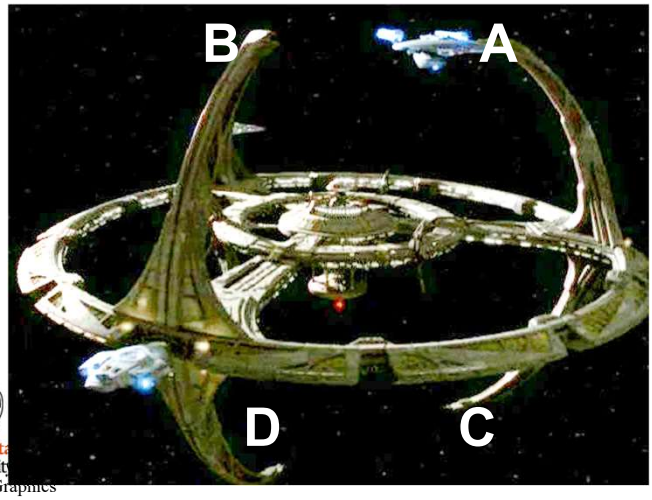



```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

Oregon State University
Computer Graphics

Did any of you ever watch *Star Trek: Deep Space Nine*?

It was about life aboard a space station. Ships docked at Deep Space Nine to unload cargo and pick up supplies. When a ship was docked at docking port "**A**", for instance, the supply-loaders didn't need to know what ship it was. They could just be told, "send these supplies out docking port A", and "bring this cargo in from docking port A".



Surprisingly, this actually has something to do with computer graphics! ☺

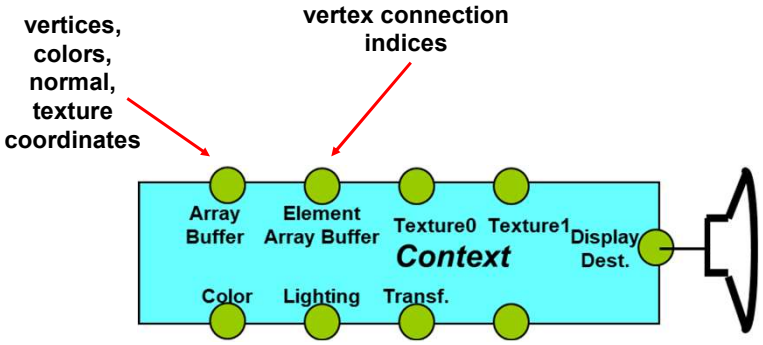
Oregon State University
Computer Graphics

The OpenGL *Rendering Context*

5

The OpenGL **Rendering Context** (also called "the **state**") contains all the characteristic information necessary to produce an image from geometry. This includes the current transformation, color, lighting, textures, where to send the display, etc.

Each window (e.g., glutCreateWindow) has its own rendering context.



Oregon State University
Computer Graphics

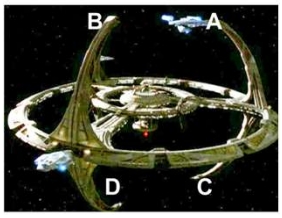
mjb – August 30, 2024

5

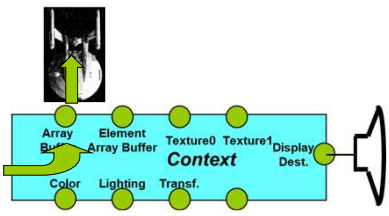
More Background – "Binding" to the Context

6

The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow in" through the Context into the object.



Vertex Buffer Object pointed to by *bufA*




```
glBindBuffer( GL_ARRAY_BUFFER, bufA);

glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
```

Oregon State University
Computer Graphics

Ships docked at Deep Space Nine to unload cargo and pick up supplies. When a ship was docked at docking port "**A**", for instance, the supply-loaders didn't need to know what ship it was. They could just be told, "send these supplies out docking port A", and "pick up this cargo from docking port A".

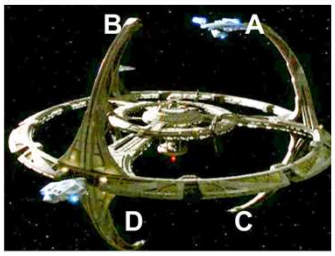
mjb – August 30, 2024

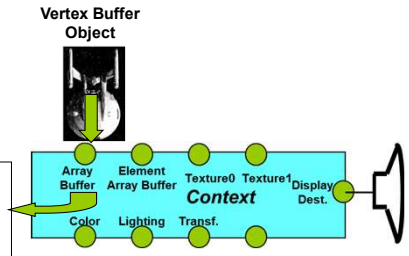
6

More Background – "Binding" to the Context

7

When you want to *use* that Vertex Buffer Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again. Its contents will "flow out" of the object into the Context.






```
glBindBuffer( GL_ARRAY_BUFFER, bufA);

glDrawArrays( GL_TRIANGLES, 0, numVertices );
```

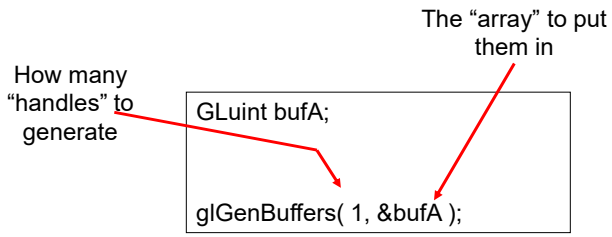
Oregon State University
Computer Graphics

More Background – How do you Create an OpenGL "Buffer Object"?

8

When creating data structures in C++, objects are pointed to by their addresses.

In OpenGL, objects are pointed to by an unsigned integer "handle". You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:



OpenGL then uses these handles to determine the actual GPU memory addresses to use.

Oregon State University
Computer Graphics

Loading data into the currently-bound Vertex Buffer Object

glBufferData(type, numBytes, data, usage);

type is the type of buffer object this is:

Use **GL_ARRAY_BUFFER** to store floating point vertices, normals, colors, and texture coordinates

numBytes is the number of bytes to store all together. It's not the number of numbers, not the number of coordinates, not the number of vertices, but the number of ***bytes***!

data is the memory address of (i.e., pointer to) the data to be transferred from CPU memory to the graphics memory. (This is allowed to be NULL, indicating that you will transfer the data over later.)

Oregon State University
Computer Graphics

Loading data into the currently-bound Vertex Buffer Object

glBufferData(type, numBytes, data, usage);

usage is a hint as to how the data will be used: GL_xxx_yyy

where xxx can be:

STATIC	this buffer will be re-written seldom
DYNAMIC	this buffer will be re-written often

and yyy can be:

DRAW	this buffer will be used for drawing
READ	this buffer will be copied into

For what we are doing, use **GL_STATIC_DRAW**

Oregon State University
Computer Graphics

11

Step #1 – Fill the C/C++ Arrays with Drawing Data (vertices, colors, …)

```
GLfloat  Vertices[  ][3] =
{
        { 1.,  2.,  3. },
        { 4.,  5.,  6. },
        . . .
};
```

Step #2 – Transfer the Drawing Data

```
glGenBuffers( 1, &bufA );

glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, Vertices, GL_STATIC_DRAW );
```

Oregon State University
Computer Graphics

mjb – August 30, 2024

11

12

Step #3 – Activate the Drawing Data Types That You Are Using

glEnableClientState(type);

where *type* can be any of:

GL_VERTEX_ARRAY
GL_COLOR_ARRAY
GL_NORMAL_ARRAY
GL_TEXTURE_COORD_ARRAY

- Call this as many times as you need to enable all the drawing data types that you are using.
- To deactivate a type, call:

glDisableClientState(type);

Oregon State University
Computer Graphics

mjb – August 30, 2024

12

13

Step #4 – To start the drawing process, bind the Buffer that holds the Drawing Data

```
glBindBuffer( GL_ARRAY_BUFFER, bufA );
```

Oregon State University
Computer Graphics

13

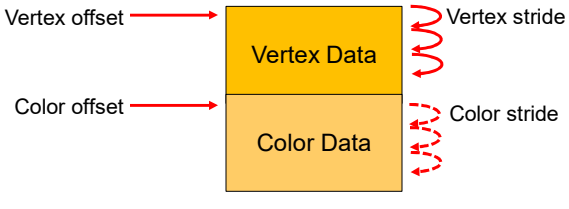
14

Step #5 – Then, specify how to get at each Data Type within that Buffer

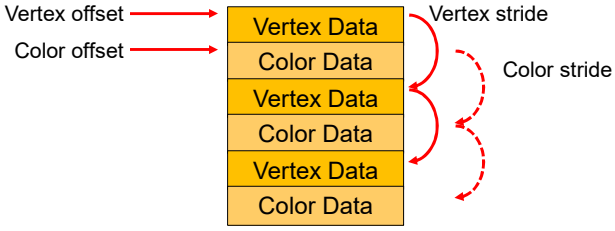
offset is the number of byte offsets from the start of the data array buffer to where the first element of this part of the data lives

stride is the number of bytes between the same type of data

Information can be stored as packed, like this:



Information can be stored as interleaved, like this:

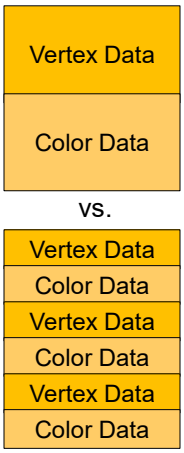


Oregon State University
Computer Graphics

14

Step #5 – Then, specify how to get at each Data Type within that Buffer


```
glVertexPointer( size, type, stride, offset);
glColorPointer( size, type, stride, offset);
glNormalPointer( type, stride, offset);
glTexCoordPointer( size, type, stride, offset);
```



size is the "how many numbers per vertex", and can be: 2, 3, or 4

type can be:

GL_SHORT
GL_INT
GL_FLOAT
GL_DOUBLE

stride is the byte offset between consecutive entries in the buffer (0 means tightly packed)

offset is the byte offset from the start of the data array buffer to where the first element of this part of the data lives.

Oregon State University
Computer Graphics

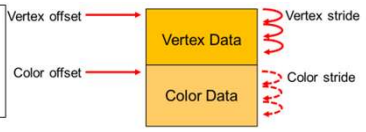
The Data Types in a vertex buffer object can be stored either as "packed" or "interleaved"

gl*Pointer(size, type, stride, offset);

Packed:

```
glVertexPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 0 );

glColorPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 3*numVertices*sizeof(GLfloat));
```

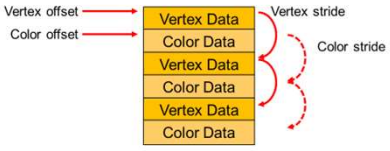
stride offset



Interleaved:

```
glVertexPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 0 );

glColorPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat) );
```

stride offset



Oregon State University
Computer Graphics

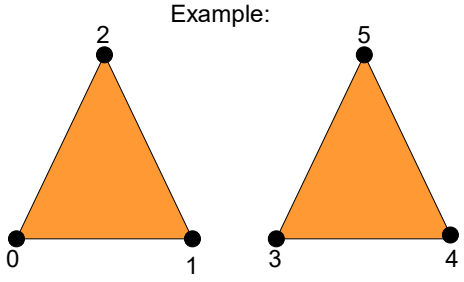
Step #6 – Draw!

17

glDrawArrays(GL_TRIANGLES, first, numVertices);

This is how you do it *if your vertices can be drawn in consecutive order*

Example:



Start with vertex #0

glDrawArrays(GL_TRIANGLES, 0, 6);

Draw 6 vertices

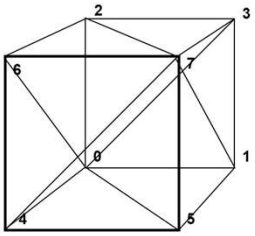
Oregon State University
Computer Graphics

mjb – August 30, 2024

17

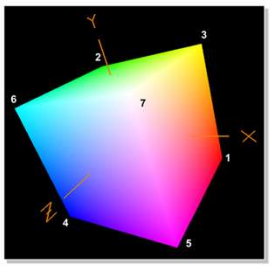
What if your vertices need to be accessed in random order?

18




```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```




```
static GLfloat CubeColors[ ][3] =
{
        { 0., 0., 0. },
        { 1., 0., 0. },
        { 0., 1., 0. },
        { 1., 1., 0. },
        { 0., 0., 1. },
        { 1., 0., 1. },
        { 0., 1., 1. },
        { 1., 1., 1. },
};
```

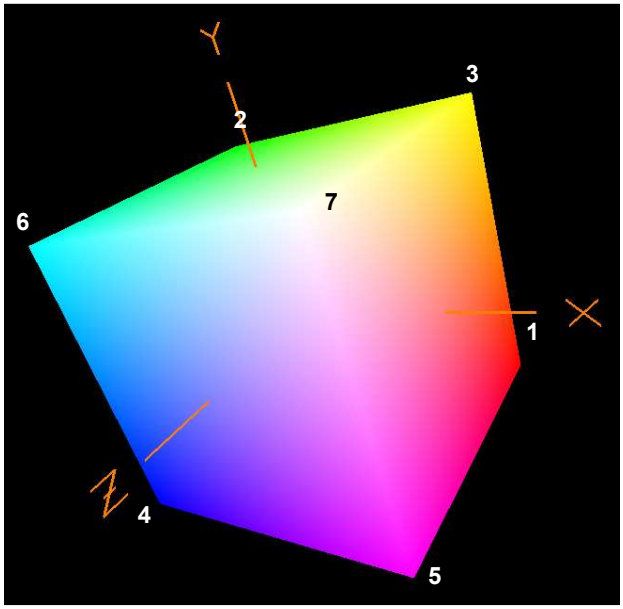
Oregon State University
Computer Graphics

mjb – August 30, 2024

18

Cube Example

19



Oregon State University
Computer Graphics

mjb – August 30, 2024

19

```
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
glVertexPointer(  3, GL_FLOAT, 0, (GLchar*) 0 );
glColorPointer(   3, GL_FLOAT,  0, (GLchar*)
                        (3*sizeof(GLfloat)*numVertices) );
glBegin( GL_TRIANGLES );
        glArrayElement( 0 );
        glArrayElement( 2 );
        glArrayElement( 3 );
        glArrayElement( 0 );
        glArrayElement( 3 );
        glArrayElement( 1 );
        glArrayElement( 4 );
        glArrayElement( 5 );
        glArrayElement( 7 );
        glArrayElement( 4 );
        glArrayElement( 7 );
        glArrayElement( 6 );
        glArrayElement( 1 );
        glArrayElement( 3 );
        glArrayElement( 7 );
        glArrayElement( 1 );
        glArrayElement( 7 );
        glArrayElement( 5 );
        . . .
```

Vertex Data

Color Data

```
        . . .
        glArrayElement( 0 );
        glArrayElement( 4 );
        glArrayElement( 6 );
        glArrayElement( 0 );
        glArrayElement( 6 );
        glArrayElement( 2 );
        glArrayElement( 2 );
        glArrayElement( 6 );
        glArrayElement( 7 );
        glArrayElement( 2 );
        glArrayElement( 7 );
        glArrayElement( 3 );
        glArrayElement( 0 );
        glArrayElement( 1 );
        glArrayElement( 5 );
        glArrayElement( 0 );
        glArrayElement( 5 );
        glArrayElement( 4 );
glEnd( );
```

```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 },
        { 0, 5, 4 }
};
```

But, this requires that all these glArrayElement( ) calls happen on the CPU and get transmitted across the bus to the GPU!

Computer Graphics

mjb – August 30, 2024

20

21

It would be better if that index array was stored over on the GPU as well

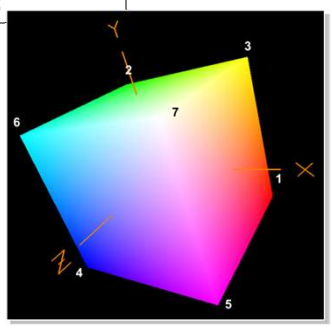
```
glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, CubeVertices, GL_STATIC_DRAW );


glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, bufB );
glBufferData( GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*numIndices, CubeTriangleIndices, GL_STATIC_DRAW );
```

Oregon State University
Computer Graphics

22

The glDrawElements( ) call

```
glBindBuffer( GL_ARRAY_BUFFER, bufA );
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, bufB );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

glVertexPointer( 3, GL_FLOAT, 0, (Gluchar*) 0 );
glColorPointer(  3,  GL_FLOAT, 0, (Gluchar*) (3*sizeof(GLfloat)*numVertices) );

glDrawElements( GL_TRIANGLES, 36, GL_UNSIGNED_INT, (Gluchar*) 0 );
```



Oregon State University
Computer Graphics

23

Writing Data into a Buffer Object, Treating it as a C/C++ Array of Structures

```
float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER,  usage );
```

usage is how the data will be accessed:

GL_READ_ONLY	the vertex data will be read from, but not written to
GL_WRITE_ONLY	the vertex data will be written to, but not read from
GL_READ_WRITE	the vertex data will be read from and written to

You can now use **vertexArray[ ]** like any other C/C++ floating-point array of structures.

When you are done, be sure to call:

```
glUnMapBuffer( GL_ARRAY_BUFFER );
```

Oregon State University
Computer Graphics

24

glMapBuffer Example

```
struct Point
{
	float x, y, z;
	float nx, ny, nz;
	float r, g, b;
	float s, t;
};

. . .

glGenBuffers( 1, &pbuffer );
glBindBuffer( GL_ARRAY_BUFFER, pbuffer );
glBufferData( GL_ARRAY_BUFFER, numPoints * sizeof(struct Point), NULL, GL_STATIC_DRAW );
struct Point * parray = (struct Point *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
for( int i = 0; i < numPoints; i++ )
{
		parray[i].x  = PointVec[i].x;
		parray[i].y  = PointVec[i].y;
		parray[i].z  = PointVec[i].z;
		parray[i].nx = PointVec[i].nx;
		parray[i].ny = PointVec[i].ny;
		parray[i].nz = PointVec[i].nz;
		parray[i].r  = PointVec[i].r;
		parray[i].g  = PointVec[i].g;
		parray[i].b  = PointVec[i].b;
		parray[i].s  = PointVec[i].s;
		parray[i].t  = PointVec[i].t;
}
glUnmapBuffer( GL_ARRAY_BUFFER );
```

Oregon State University
Computer Graphics

Using our Vertex Buffer Object C++ Class

Declaring a Global:

```
VertexBufferObject  VB ;
```

Filling:

```
VB.Init( );
VB.glBegin( GL_TRIANGLES );                    // can be any of the OpenGL topologies
for( int i = 0; i < 12; i++ )
{
        for( int j = 0; j < 3; j++ )
        {
                int k = CubeTriangleIndices[ i ][ j ];
                VB.glColor3fv( CubeColors[ k ] );
                VB.glVertex3fv( CubeVertices[ k ] );
        }
}
VB.glEnd(  );
```

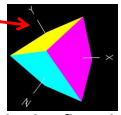
This is available from our Class Resources page

Drawing:

```
VB.Draw(  );
```

Oregon State University
Computer Graphics

Vertex Buffer Object Class Methods

void CollapseCommonVertices(bool);	*true* means to not replicate common vertices in the internal vertex table. This is good if all uses of a particular vertex will have the same normal, color, and texture coordinates, like this – instead of like this.
	 
void Draw( );	Draw the primitive. If this is the first time Draw( ) is being called, it will setup all the proper buffer objects, etc. If it is a subsequent call, then it will just initiate the drawing.
void DrawInstanced(numInstances);	Same as Draw( ), but will draw multiple instances. Must be using shaders to make this worthwhile.
void glBegin(topology);	Begin the definition of a primitive.
void glColor3f(r, g, b); void glColor3fv(rgb[3]);	Specify a vertex's color.
void glEnd( );	Terminate the definition of a primitive.

Ore
Ur
Compu

Vertex Buffer Object Class Methods

void glNormal3f(nx, ny, nz); void glNormal3fv(nxyz[3]);	Specify a vertex's normal.
void glTexCoord2f(s, t); void glTexCoord2fv(st[2]);	Specify a vertex's texture coordinates.
void glVertex3f(x, y, z); void glVertex3fv(xyz[3]);	Specify a vertex's coordinates.
void Print(char *text, FILE *);	Prints the vertex, normal, color, texture coordinate, and connection element information to a file, along with some preliminary text. If the file pointer is not given, standard error (i.e., the console) is used.
void RestartPrimitive( );	Causes the primitive to be restarted. This is useful when doing triangle strips or quad strips and you want to start another one without getting out of the current one. By doing it this way, all of the strips' vertices will end up in the same table, and you only need to have one *VertexBufferObject* class going.

Oregon State University
Computer Graphics

Notes

• If you want to print the contents of your data structure to a file (for debugging or curiosity), do this:

```
FILE *fp = fopen( "debuggingfile.txt", "w" );
if( fp == NULL )
{
        fprintf( stderr, “Cannot create file ‘debuggingfile.txt’\n” );
}
else
{
        VB.Print( "My Vertex Buffer :", fp );
        fclose( fp );
}
```

• You can call the *glBegin* method more than once. Each call will wipe out your original display information and start over from scratch. This is useful if you are interactively editing geometry, such as sculpting a curve.

Oregon State University
Computer Graphics

A Caveat

Be judicious about collapsing common vertices! The good news is that it saves space and it might increase speed some (by having to transform fewer vertices). But, the bad news is that it takes much longer to create large meshes. Here's why.

Say you have a 1,000 x 1,000 point triangle mesh, drawn as 999 triangle strips, all in the same *VertexBufferObject* class (which you can do using the *RestartPrimitive* method) .

When you draw the S^{th} triangle strip, half of those points are coincident with points in the S-1^{st} strip. But, to find those 1,000 coincident points, it must search through 1000*S points first. There is no way to tell it to only look at the last 1,000 points. Even though the search is only $O(\log_2 N)$, where N is the number of points kept so far, it still adds up to a lot of time over the course of the entire mesh.

It starts out fast, but slows down as the number of points being held increases.

If you did have a 1,000 x 1,000 mesh, it might be better to not collapse vertices at all. Or, a compromise might be to collapse vertices, but break this mesh up into 50 *VertexBufferObjects*, each of size 20 x 1,000.

Oregon State University
Computer Graphics

Just a thought…

Drawing the Cube With Collapsing Identical Vertices

```
GLuint CubeTriangleIndices[ ][3] =
{
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 }
        { 0, 5, 4 }
};
```

Define 8 points

X	Y	Z
-1.00	-1.00	-1.00
-1.00	1.00	-1.00
1.00	1.00	-1.00
1.00	-1.00	-1.00
-1.00	-1.00	1.00
1.00	-1.00	1.00
1.00	1.00	1.00
-1.00	1.00	1.00

Define 8 colors

R	G	B
0.00	0.00	0.00
0.00	1.00	0.00
1.00	1.00	0.00
1.00	0.00	0.00
0.00	0.00	1.00
1.00	0.00	1.00
1.00	1.00	1.00
0.00	1.00	1.00

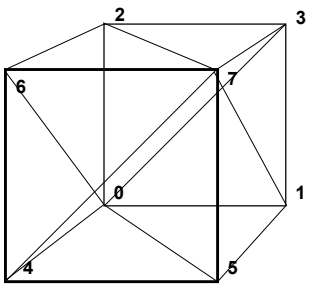
Draw 36 array indices:

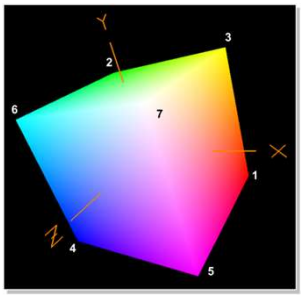
0	2	3
0	3	1
4	5	7
4	7	6
1	3	7
1	7	5
0	4	6
0	6	2
2	6	7
2	7	3
0	1	5
0	5	4

Oregon State University
Computer Graphics

Drawing the Cube Without Collapsing Identical Vertices

31





Define 36 vertices and 36 colors

X	Y	Z	R	G	B
-1.	-1.	-1.	0.	0.	0.
	. . .			. . .	

Using Vertex Buffers with Shaders

32

Let's say that we have the following **vertex shader** and we want to supply the vertices from a Vertex Buffer Object.

```
in vec3 aVertex;
in vec3 aColor;

out vec3 vColor;

void
main( )
{
        vColor = aColor;
        gl_Position = gl_ModelViewProjectionMatrix * vec4( aVertex, 1. );
}
```

Let's also say that, at some time, we want to supply the colors from a Vertex Buffer Object as well, but for right now, the color will be uniform.

Using Vertex Buffers with Shaders

We're assuming here that
- we already have the shader program setup in *program*
- we already have the vertices in the *vertexBuffer*

```
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY  );

GLuint vertexLocation = glGetAttribLocation( program, "aVertex" );
GLuint colorLocation  = glGetAttribLocation( program, "aColor" );

glVertexAttribPointer( vertexLocation, 3, GL_FLOAT, GL_FALSE, 0, (GLuchar *)0 );
glEnableVertexAttribArray( vertexLocation );          // dynamic attribute

glVertexAttrib3f( colorLocation,  r, g, b );          // static attribute
glVertexAttribArray( colorLocation );

glDrawArrays( GL_TRIANGLES,  0,  3*numTris );
```

Oregon State University
Computer Graphics

Using Vertex Buffers with the Shaders C++ Class

We're assuming here that
- we already have the vertices in the *vertexBuffer*
- we have already created a C++ GLSLProgram class object called *Pattern*

```
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY  );

Pattern.SetVertexAttributePointer3fv( "aVertex", (GLfloat *)0 );
Pattern.EnableVertexAttribArray( "aVertex" );          // dynamic attribute

Pattern.SetVertexAttributeVariable( "aColor",  r, g, b );  // static attribute
Pattern.EnableVertexAttribArray( "aColor" );

glDrawArrays( GL_TRIANGLES,  0,  3*numTris );
```

Oregon State University
Computer Graphics