CITS3003

Daniel Paparo

1 Lecture 1

1.1 Elements of image formation

- Object
- Viewer
- Lighting

Object - Exists in space independant of any viewer or image process. Is a set of locations in space.
Viewer - The viewer forms the image of the objects.
Light - Causes a reaction in the visual system. An ideal light source emits in all directions equally. A light source is characterized by the way it emits each light frequyencies and at what intensities.

Luminance image - Monochromatic greyscale, with values in grey levels.
Field of view - Maximum angle that can be imaged by the camera.

1.2 Color models

Additive color model - RGB color mixing, in which every color is a result of red, green and blue being added together in some combination.
Subtractive color model - CMY color mixing - Arises from cyan, magenta and yellow being subtracted away from eachother in combinations.

1.3 Synthetic camera model

OpenGL's synthetic camera is based on a pinhole camera. While in a real camera the image comes out inverted, the image in openGL comes out correctly.

- Ray tracing is not used in openGL as it can be too slow on standard user's hardware
- Platform independent API

2 Lecture 2

2.1 OpenGL

- Allows computer programs to achieve fast graphics performance by using the GPU rather than CPU
- Allows applications to control the GPU through programs known as shaders
- It is the applications job to send data to the GPU
- GPU does all rendering

Shaders - In charge of sending data through to the GPU.

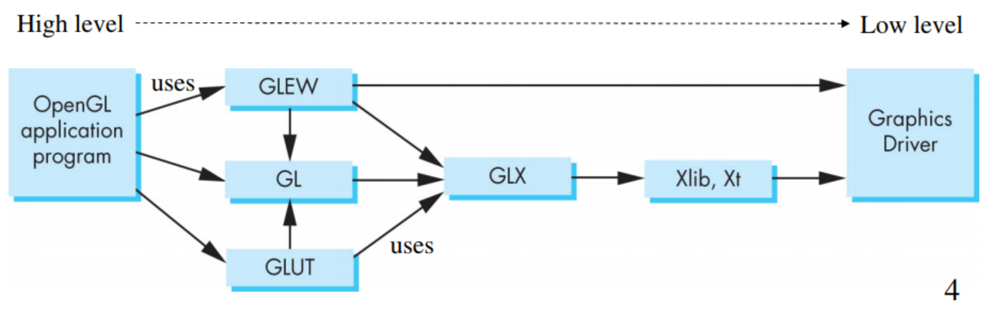
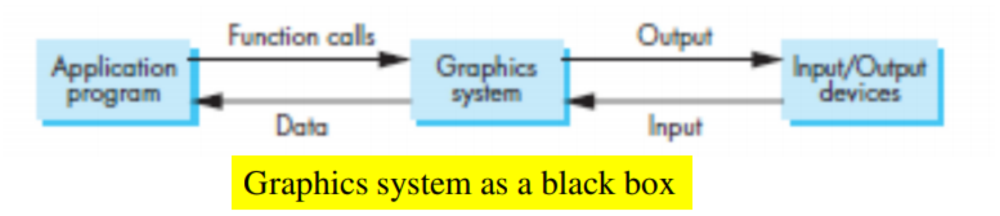


Diagram of low to high level GL



State Machine interactions

2.1.1 Software organization

The applications can use GLEW, GL, GLUT functions but not directly access GLX, XLIB, etc. The program can be recompiled with GLUT libraries for other operating systems without extra effort - This adds portability!

2.1.2 OpenGL Types

In OpenGL we need to use GL datatypes instead of C types, but under the same process

- GLfloat
- GLdouble
- GLint

2.1.3 OpenGL as a State Machine

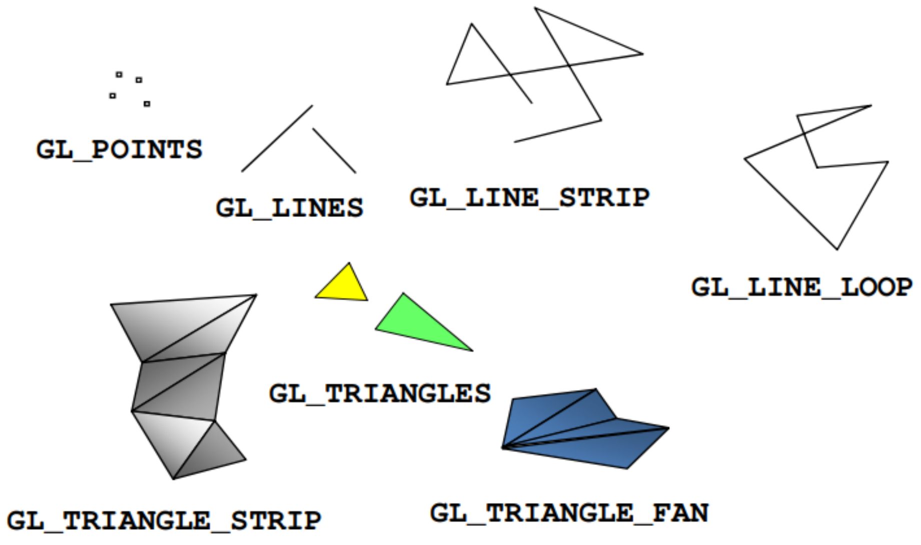
We can think of the entire graphics system as a *state machine*.

State machine - a device which can be in one of a set number of stable conditions depending on its previous condition and on the present values of its inputs.

The state machine has inputs coming from the application program. These inputs can change the state of the machine or can cause the machine to produce a visable output.

From this perspective of the API there are two types of graphics functions:

1. Functions that define primatives that flow through the state machine:
    - Define how vertices are processed
    - Can cause output if the primative is visable
2. Functions that either change the state inside the machine or return the state information



OpenGL Primatives

- Tranformation functions
- Attribute functions
- In OpenGL, most state variables are defined by the application and sent to the shaders

2.2 OpenGL functions

The openGL functions provide a wide range of functions specifying: primatives(ie. Points, line segments and triangles), attributes, tranformations (ie. Viewing and modelling), Control(GLUT), input(GLUT), and query.

- Open GL is restricted to triangles

2.2.1 OpenGL Primatives

2.2.2 Attributes

Attributes are properties associated with the primitives that give them their different appearance:

- Color (for points, lines, polygons)
- Size and width (for points, lines)
- Stipple pattern (for lines, polygons)
- Polygon mode
  - Display as filled: solid color or stipple pattern
  - Display edges
  - Display vertices

2.2.3 Lack of object orientation

OpenGL is not object oriented as a matter of increasing speed. It is possible to overload function in C++ to take advantage of objects but this creates an issue with efficiency.

2.2.4 OpenGL functions

```
glUniform3f(x,y,z);
```

Where:

- `gl` refers to the GL library
- `Uniform` is the function name
- `3` is the dimensions
- `f` refers to x,y and z being floats

The above example, `glUniform` specify the value of a uniform variable for the current program object.

```
glUniform3fv(x,y,z);
```

Where:

- `v` refers to an array of floats

2.2.5 OpenGL #Defines

`#include <GL/glut.h>` Will include most other necessary libraries.

2.3 GLSL

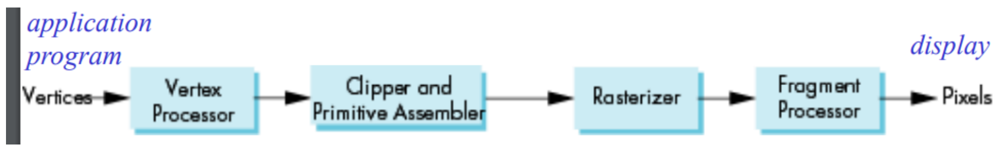
GLSL is shor for "OpenGL Shading Language". It is a C-like language with built in matriox and vector types, overloaded operators and C++ like constructors.

2.4 A simple program

An OpenGL program will consist of:

- `main()` : Creates the window, calls the `init()` function, specifies callback functions (associated with some event) relavent to the application, and enters an event loop.
- `init()` : Defines the vertices, attributes, etc. of the objects to be rendered, specifies the shader programs.
- `display()` : this is a callback function that defines what to draw whenever the window is refreshed

```
#include <GL/glut.g>

void init(){
    //Code to be inserted here
}
```



Development Pipeline

```
void mydisplay(){
    glClear(GL_COLOR_BUGGER_BIT);
    //need to fill in this part and add in shaders
}

void main(int argc, char**argv){
    glutCreateWindow("simple");  //Creates a window with "simple" as its title
    init();                      //Set up OpenGL state and initialize shaders
    glutDisplayFunc(mydisplay);
    glutMainLoop();              //Enter event loop
}
```

3 Lecture 3

3.1 Pipeline archetecture

Throughput: Amount of data which can enter or exit at one time
Latency: The time it takes for one bit to go from start of pipeline to endd of pipeline.
There must be a balance between throughput and latency.

3.2 Graphics piplines

3.2.1 Vertex processing

Work of the pipeline is to convert from one co-ordinate system to another: * Object, camera and screen coordinates

3.2.2 Projection

Combines 3D objects with 3D viewer, to produce a 2D image - Must be projected in a way which can be displayed on a screen.
Perspective projection: all projected rays meet at the center of projection
Parallel projection: projected rays are parallel, centre of projection is at infinity. (In OpenGL, specify the direction of projection instead of the centre of projection)

3.2.3 Primative assembly

Vertices must be collected into geometric objects before clipping and rasterization can take place.

3.2.4 Clipping

Camera cannot see the full image (the field of view), therefore some of the image outside of this must be clipped. This area which is in view is refferred to as the **View Volume**.

3.2.5 Rasterization

Objects that are not clipped out will have appropriate poixels in the frame buffer and must be assigned colors. Rasterizing produces a set of **fragments**. Fragments are potential pixels which have a location, colour, depth and alpha attributes.

3.2.6 Fragment processing

To determine the color of the corresponding pixel.

3.2.7 Immediate mode graphics

- Objects specified by vertices
- Older versions of openGL use this mode
- Nothing is stored in the CPU or GPU

Advantages

- No memory is required

Disadvantages

- Cannot be restored without being rerendered through the complete timeline
- Graphics bottleneck between the CPU and GPU

3.2.8 Retained mode graphics

- Put all data in an array
- Send all data to be rendered in the GPU
- Stores output of the process
- If you want to change anything all you need to do is send the GPU a change
- Requires an increase in memory, but an increase in GPU/CPU efficiency

3.3 Polygon issues

OpenGL will only display triangles:

- Simple: edges cannot cross
- Convex: all points on a line segment between two points in a poly are also a poly
- Flat: all vertices are in the same place

Triangulation/Tessalation: Converting a polygon to triangles
Equilateral triangles render well. Triangles with long and thin angles will cause issues. We need to maximize the miknimum most interior angle for maximum efficiency.

3.4 Color

3.4.1 RGB Color

Each color component is stored seperately in the frame buffer

3.4.2 Indexed colors

An indexed list of RGB colors (8-bit) which can be used to save space when creating a program with standard colors

3.5 Shading:

Smooth Shading: Interpolates vertex colors across visable polygons.
Flat Shading: Color of the first vertex determines the fill color.

4 Lecture 4

OpenGL is about displaying objects on the screen. In openGL programs we define an object by:

1. Putting all its vertices geometric data in an array (retained mode graphics system):

```
vec3 points[3];
points[0] = vec3(0.0, 0.0, 0.0);
points[1] = vec3(0.0, 1.0, 0.0);
points[2] = vec3(0.0, 0.0, 1.0);
```

2. Send the array to GPU
3. Tell the GPU to render the points as a triangle (for this example).

The GPU's memory is about how much data it can hold of these arrays of triangles.

4.1 OpenGL program structure

- `main()` - more complex than before; Mostly calling GLUT functions
- `init()` - Will incorporate color
- `initShader()` - details of setting up shaders will be looked at in later lectures
- Key issue is that we must form a data array to send to the GPU then render it

4.2 GLUT Functions

`glutInit`: Initializes the GLUT system and allows it to recieve command line arguments
`glutInitDisplayMode`: Requests properties for the window (RGBA color or indexed color, Double Buffering, or Depth Buffer). The function uses the bitwise ORed together with `|` to say that both options are wanted.
`glutWindowSize`: Defines the window size in pixels
`glutWindowPosition`: Positions the window
`glutCreateWindow`: Creates window with some title
`glutDisplayFunc`: Sets the display callback

`glutKeyboardFunc`: Sets the window resize callback
`glutReshapeFunc`: Sets the window resize callback
`glutTimerFunc`: Sets the timer callback
`glutIdleFunc`: sets the idle callback
`glutMainLoop`: enters infinite event loop - Note that this will never return but may exit.

4.3 Display Callback

Once we get data to the GPU, we can initiate the rendering with a simple display

```
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArray(GL_TRIANGLES, 0, 3);

    glutSwapBuffers();
}
```

4.4 Initialization

All the intialization codes can be put inside an `init()` function.

4.5 Vertex arrays

Vertices can have many attributes including position, color etc. The vertex array holds all of this data, these are of type `vec2` or `vec3`.

4.6 Vertex Buffer Objects

Vertex buffer objects: allow us to tranfer large amounts of data to the GPU.

This is how the data in the current vertex array makes it to the GPU. Different attriubtes, such as points and colors, can be stored in this buffer. Multiple buffers can be assigned.

4.7 Reading, compiling and linking shaders

The function `InitShader` defined in `InitShader.cpp` Carries out the reading, compiling and linking of the shaders. If there are errors in any of the glsl file, the program will crash at this line

4.8 Coordinate frames

In OpenGL there are 6 coordinate frames specified in the pipeline:

1. Object coordinates - The units of the points
2. World Coordinates - The virtual world
3. Camera coordinates - Where the camera is (normally kept the same as the world coordinates)
4. Clip Coordinates - Objects that are not inside the view volume are clipped out.
5. Normalized device coordinates - After perspective division fives 3D representation
6. Window coordinates - Calculating which pixel gets what color

4.9 The OpenGL camera

OpenGL places a camera at the origin of the object coordinate space pointing in the negative z direction. The default viewing volume is a box centered at the origin with a width of two.

Orthographic viewing: In the default orthographic view, points are proejcted forwared along the z axis into the plane $z = 0$

4.10 Viewports

Where should everything be displayed? We do not have to use the entire window to render the scene. Viewports are simply to use part of the window - they can be changed in dimensions etc.

4.11 Transformation and viewing

In openGL, the projection is carried out by a projection matrix. Tranformations are used for changes in the coordinate systems.

5 Lecture 5

Vertex shaders: Provide final transformations of mech vertices.
Fragment shaders: A Fragment Shader is the Shader stage that will process a Fragment generated by the Rasterization into a set of colors and a single depth value.

There are no pointers in GLSL.

5.1 GLSL Qualifiers

- Storage quantifiers
- Precision qualifiers
- Parameter qualifiers

5.2 Storage qualifiers

Qualifier constant is used in the same way as java. The variable is read only.

Qualifier attribute: Not changable.

5.2.1 The qualifier uniform

Global in scope, are used to describe global properties such as light position, materials etc. of the object. They are similar to the quantifier attribute. They are similar to the quantifier attribute. They can only be used with the floating point scale.

5.3 Precision qualifiers

Can affect power usage and rendering quality - a trade off must be made to optimize the result

5.4 Parameter qualifier

- Qualifieirs in: Read only
- Qualifiers out: Write only
- Qualifiers in-out: Read and write

5.5 Built in variables

- `gl_Position`: Position passed to the rasterizer - must be the output of all shaders
- `gl_FragColor`: Each fragment will output a color for the fragment

5.6 Swizzling and selection

Swizzling: Lets us manipulate components easily.
Selection: Using the "." operator

6 Lecture 6

Note: must be familiar with the graphics pipeline for the final exam

Vertex shader: The Vertex Shader is the programmable Shader stage in the rendering pipeline that handles the processing of individual vertices.
Geometric transformations: Occur in the vertex shader and change relative location rotation scale etc.
Moving Vertices: Perform morphing, such as computing motion and particle simulation.
Lighting: Calculate shading using light and surface properties. Calculate cartoon shading too.

In the rendering pipeline each vertex is manupulated/processed independently - this is through the vertex shader: It takes one vertex from the vertex stream and generates the transformed vertex to the output vertex stream.

Shaders: Small programs which are compiled and run on the GPU. They can be run in parallel to render extremel complex scenes.

6.1 Simple vertex shader

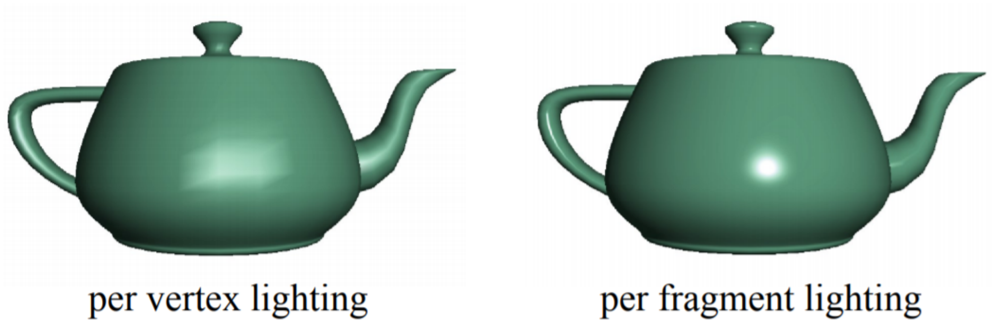
```
#version 150
in vec4 vPosition;

void main(void)
{
    gl_Position = vPosition;
}
```

The vertex shader is essentially for generatign data to be outputted to the data shader. Color of vertices is extremely important for blending. The shader can be accessed by the program but it is more meant to be used as a black box which should not be looked into

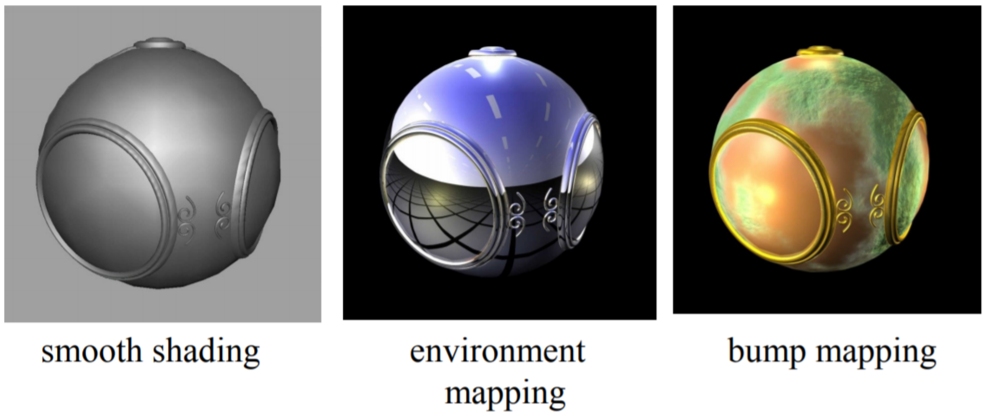
6.2 Fragment shader

6.2.1 Per fragment lighting

Fragment: Potential pixels with color, depth etc.



Per fragment lighting



Fragment shader applications

6.2.2 Fragment shader applications

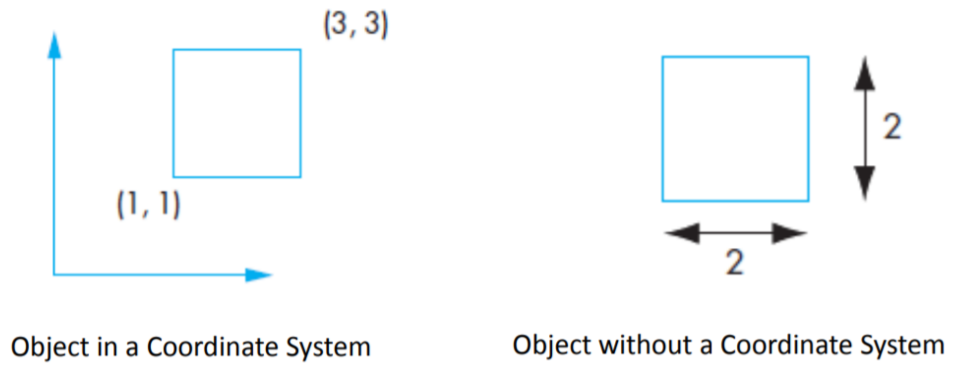
Texture mapping can also be done at the fragment level.
Lighting calculations: Per fragment calculations.
Texture mapping: Bump mapping, enviromental mapping.

1. Vertex data
2. Vertex shader
3. Fragment shader
4. Final color

6.3 Unity between shaders

The shaders and application programs must work together

- All attribute's variable names are stored in a table
- The application can search for an index of the attribute variable
- Same is true for a uniform


(3, 3)
(1, 1)
Object in a Coordinate System
2
2
Object without a Coordinate System

Coordinate free geometry

7 Lecture 7

7.1 Geometric objects

Points: A location in soace without size or shape
Scalar: A $\mathbb{R}$ or $\mathbb{C}$ number used to specify some quantity
Vectors: A line segment that allows us to work with lines or dimensions, but has no fixed location in space.

7.2 Coordinate free geometry

7.3 Spaces

Scalar spaces:

- Scalars can be combined to form another
- Add and subtract can be applied to it

Vector spaces:

- Contains vectors and scalars

Euclidean space:

- An extension of a vector space that adds a measure of size and distance

Affine Space:

- has scalars, vectors and includes points

7.3.1 Dimension of a space

The number of vectors which are linear will determine the dimension of the space - these vectors form the basis.

7.3.2 Coordinate system

The coordinate system creates some perspective to find the location of some object in the space

Change of coordinates Each basis of any coordinate system can be expressed in terms of another coordinant system.

8 Lecture 8

8.1 Coordinate frames

Basis vectores alone cannot represent points as there must be some origin point - a coodinate frame must contain some origin for it to make sense.

A coordinate frame is represented as:

$$(\vec{P_0}, \vec{v_1}, \vec{v_2}, \vec{v_3})$$

where every vector is expressed as:

$$\vec{v} = \alpha\vec{v_1} + \alpha\vec{v_2}\alpha\vec{v_3}$$

and every point can be expressed as:

$$\vec{P} = \vec{P_0} + \beta_1\vec{v_1} + \beta_2\vec{v_2} + \beta_3\vec{v_3}$$

This will be sufficient to define some point

8.2 Homogenous coordinates

Since $0 \times \vec{P} = 0$ and $1 \times \vec{P} = \vec{P}$, a vector can be expressed as:

$$\vec{v} = [\alpha_1, \alpha_2, \alpha_3, 0]$$

and a point as:

$$\vec{P} = [\beta_1, \beta_2, \beta_3, 1]$$

This is a much nicer representation of the vectors, which means they can be treated similarly as they are both of the same size. Homogenous coordinates are so important because it makes all of the vectors standard, which allows for the square matrices to be changed from coordinate systems. Hardware pipelines are designed to work with these 4-Dimensional vectors.

8.3 Common transformations

8.3.1 Rigid transformations

$$\begin{bmatrix} R & t \\ \varnothing^T & 1 \end{bmatrix}$$

- Preserves everything
- 6 degrees of freedom

8.3.2 Similarity transformation

$$\begin{bmatrix} sR & t \\ \emptyset^T & 1 \end{bmatrix} or \begin{bmatrix} R & t \\ \emptyset^t & s' \end{bmatrix}$$

- Preserves the angle, and ratios of length and area
- Scales up and scales down
- 7 Degrees of freedom

8.3.3 Affine transformations

$$\begin{bmatrix} A & t \\ \emptyset^t & 1 \end{bmatrix}$$

- Preserves parralellism, ratios of length and area
- 12 degrees of freedom

8.3.4 Persepective transformation

- Preserves cross ratios
- Makes near objects seem bigger the furthest
- Preserves ratios of ratios of height

8.4 World and camera coordinate frames

- Points are defined as 4×4 arrays
- OpenGL will operate in 4×4 matrix calculations

We start in the world frame and then change into camera frame

8.4.1 Moving the camera

The camera or world can be moved

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix shows an amount of $-d$ in the z axis, from the standard 4×4 identity matrix.

8.4.2 General transformations

Transformations: Maps points onto other points

8.5 Notation

$\vec{P}, \vec{Q}, \vec{R}$: denote a point
$\vec{p}, \vec{q}, \vec{r}$: denote a 4D point
$\vec{u}, \vec{v}, \vec{w}$: Vector in affine space
α, β, γ: define scalars

9 Lecture 9

9.1 Transformations

Transformation: A function which takes a point or vector and maps it onto some other point or vector

The difference between a transformation and a change of frame is that: if C is non-singular, then each linear transformation is a change of frame.

Transformation:

$$\vec{v} = \vec{C}\vec{u}$$

Change of frame:

$$\vec{u} = \vec{M}\vec{u}$$

Every linear transformation is either:

- A change of frame, or
- A change of the vertices in teh same frame

9.1.1 Translations

The act of moving fixed points by a set displacement by adding some vector.

$$P' = P + d$$

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

9.1.2 Rotation

This is to transform some point by some given angle

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

The rotation transformation features:

- A fixed point
- reinterpretation from 2D to 3D
  - Anticlockwise is posetive
- Rotation of 2D is equivilant to rotating the 3D plane about the z axis.

$$P' = SP$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D rotation

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

9.1.3 Ridgid vs non-ridgid body transformations

Ridgid: The points within the object do not change with respect to eachother
Non-ridgid: The points within the object do not change with respect to eachother

9.1.4 Scaling

Uniform scaling for maintaining ratios between points - non-uniform for not maintaining the ratios.

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

9.1.5 Reflection

Corresponds to the negative scale factors:

10 Lecture 10

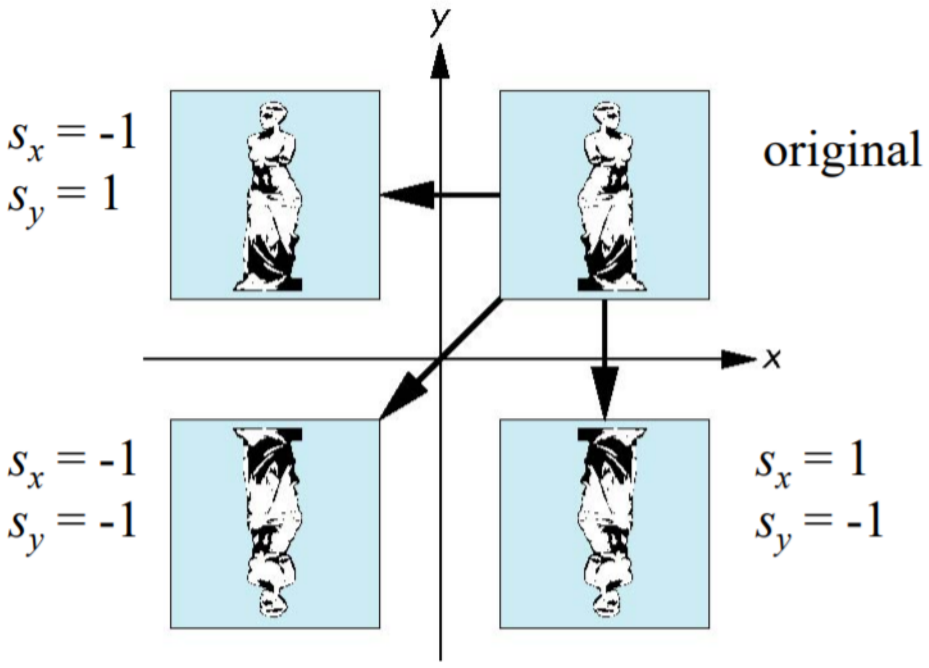
10.1 Interactive paradigm

1. User sees an object on the display
2. User points to the object with an input device
3. Object changes
4. Repeat

10.2 Logicical input devices

Based on your user input the program reacts in some way and changes are displayed. User input devices control the program, these devices can be:

- Physical devices, which are defined by their physical characteristics.



Reflection

- Logical devices, which are defined by their interactions.

Logical devices are the interaction type, which can come from different physical devices - for example both a keyboard and a mouse can select text.

We cannot tell the physical input from our programs (can be a keyboard input or could be a mouse input), but what the code can give us is the logical inputs. Older graphics API's provide six logical inputs:

- **Locator**: Returns a location
- **Choice**: Returns one item out of n number of options
- **String**: Returns a string of characters
- **Stroke**: Returns an array of positions
- **Valuator**: Returns a floating point number
- **Pick**: Returns the ID of some object

OpenGL and GLUT accept all of these logical inputs.

10.2.1 Incremental devices

While a data tablet provides definite x and y values for the location of the pointer input, a mouse or joystic will only allow for relative changes to be sent (for example: in one cycle the mouse moved 3cm upwards). The PC must integrate these to find an absolute position. It may be necessary to remember the previous absolute position.

10.2.2 Input modes

Input devices have a trigger which is used to send a value to the operating system:

- Button on the mouse
- Pressing or releasing a key

When the device is triggered it will send a signal to the system (such as a position or unicode character). An input mode concerns how and why the device was triggererd.

Request mode: Only sent to program when the user triggers the device (i.e. a keyboard click). Program will hang until the input is given. This mode is not suitable for interactive programs.
Event mode: The program specifies a number of events which are of interest and will react accordingly when it arises. This is done by using an event loop.

10.3 Events

Window event: Resizing, exposing or reconfigurating the window.
Mouse event: Clicking the mouse buttons
Motion events: Also known as a mouse move event
Keyboard events: A key is pressed
Idle event: No input at all (this is dealt with by an `idle function`)

10.4 Callbacks

Callback functions can be defined for each type of event which can occur:

```
void myMouseFunction(int button, int state, int x, int y){
    //Some action
}

glutMouseFunc(myMouseFunction);
```

This function, `myMouseFunction()` will be called when some mouse event occurs.

10.4.1 GLUT event loop

The last line of the programs main function must be:

```
GlutMainLoop();
```

Which puts the program into an infinite event loop

10.4.2 Display callback

The display callback is run whenever the indow is supposed to be refreshed, for example: When the window is opened, reshaped exposed or the pgram wants to change something. In the main function:

```
glutDisplayFunc(myDisplayFunction);
```

Some events may ask for multiple display callbacks in a single loop, this can be solved by using:

```
glutPostRedisplay();
```

Which flags an iteration to be refreshed after an event. Glut checks this after each iteration needing to be refreshed, which can save on efficiency.

11 Lecture 11

11.1 Input modes

Request Mode
Event mode
Callback function: A function tat handles events
Windowing system: Multiple viewports wihch can be opened concurrently
In openGL the window coordinate system is three dimensional.

- In raster image systems origin is at point $(0, 0)$ - top left
- In OpenGL This position is bottom left

11.2 Display resolution

Display resolution can change, and as such the frame buffer must be of atleast the size of the display.

11.3 Events

- Move event
- Mouse event

To obtain the window size:
First we must find the height of the window:

- We can define it as a constant
- We can use the function to tell us what it is.

11.3.1 Terminating the program

A keyboard function can be used to terminate the program in the following example:

```
void mykey(unsigned char key, int x, int y)
{
    if (key == 'Q' | key == 'q')
    exit(0);
}
```

glutPostRedisplay(): Sets a flag for the screen to be redisplayed (refreshed) at the end of an iteration - this avoids doing multiple redisplays in a single iteration.

11.3.2 Reshaping the window

The window can be shaped by resizing the window from the sides (supported in most operating systems). This can be done according to three main methods:

1. Force fit (Changing the aspect ratio)
2. Force to keep all sizes the same (causes clipping)

3. Force display at same aspect (makes image smaller in the window - will occupy the space according to some square surrounding the original image).

```
void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    //glOrtho(left,right,bottom,top,near,far)
    glOrtho(-0.2*(float)w/(float)h, 0.2*(float)w/(float)h, -0.2, 0.2, 2.0, 20.0);
}
```

Idle callback: is invoked when there is nothing else to do at that point in time - will often simply just set a redisplay flag.

11.3.3 Menu items

OpenGL has support for popup dropdown menus. In order to create a menu the following must be done:

1. Define the menu
2. Assign actions to each menu item.
3. Attatch to some mouse button

```
menu_id = glutCreateMenu(mymenu);
glutAddmenuEntry(”clear Screen”, 1);
gluAddMenuEntry(”exit”, 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Adding a sub-menu can be done using:

```
glutAddSubMenu(char *submenuName, int submenuId);
```

And a menu call back function will need to be defined in the following way:

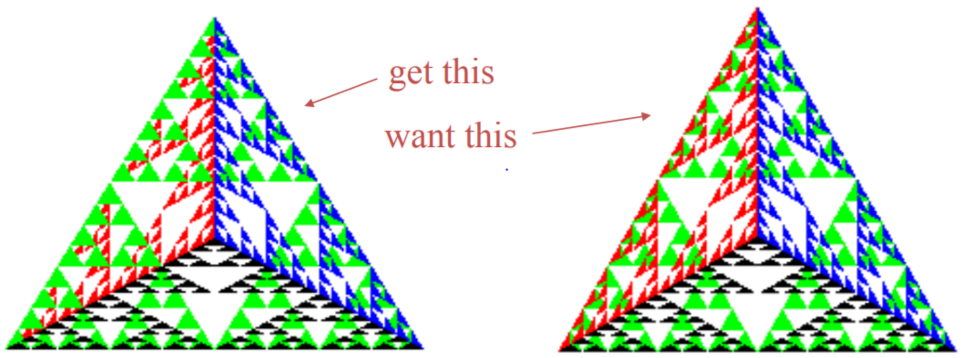
```
void mymenu(int id)
{
    if (id == 1) glClear();
    if (id == 2) exit(0);
}
```

12 Lecture 12

Not relavent - Project information

13 Lecture 13

OpenGL is specifically designed for 3D graphics. To switch to a 2D system we must simply use `vec3`, `glUniform3d` and there is no need to do hidden surface removal as they are all on the same plane.



Wrong faces are at from

13.1 The sierpinski gasket

Every time a division is occuring: the area decreases and the perimeter increases in the following equations::

$$Area = (\frac{3}{4})^n \frac{LH}{2}$$

Which shows that as $n \to \infty, A \to 0$

$$Perimeter = (\frac{3}{2})^n 3L$$

Which shows that as $n \to \infty, P \to \infty$

This is what is described as a fractal - It is no ordinary geometric object - It is neither two, nor three dimensional - it is instead a fractal object.

In a 3D sierpinki's gasket instead of dividing 3 sides we are subdividing 4 faces into smaller tetrahydra.

13.2 Hidden surface removal

In rendering the 3D sierpinski gasket openGL will render the faces in order that they come to the fragment shader, often resuting in faces being displayed in the wrong order.

This shows that there is a requirement for the system to know which triangles should be displayed and what to hide - **Hidden surface removal**.

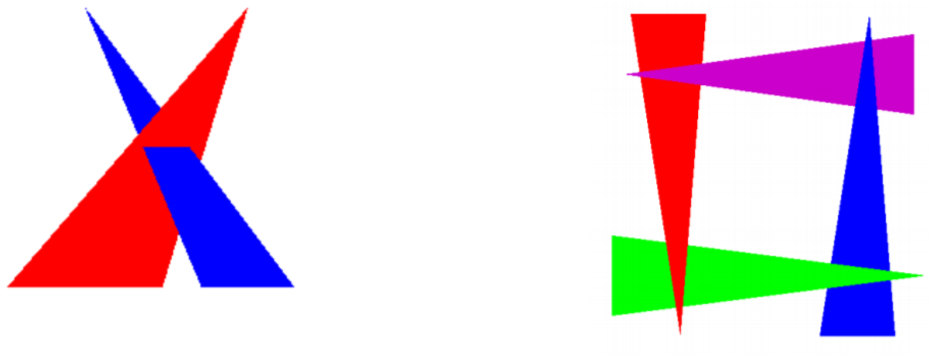
Hidden surface removal: Removing surfaces which are obscured by other faces

OpenGL uses the **z buffer algorithm** to determine which faces will and wont be displayed. This holds data the entire way through the processings pipeline of the z indexes (for ease involving transparency), and will have the same resolution of the final image.

13.2.1 Z buffer algorithm

The most widely used and simplest hidden frame removal system - can be immplemented in either the hardware or software - Must be perfomed framentwise - works in the "image space".

- Computers a color for each point of intersection (p)
- Must check whether p is visable, if it is closer to the camera than another polygon point



Pitfalls of painters algorithm

```
for each pixel (i,j) do
    Z-buffer[i,j] ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each polygon A do
    for each pixel (i,j) occupied by A do
        Compute depth z and shade s of A at (i,j)
        if z > Z-buffer[i,j] then
            Z-buffer[i,j] ← z
            Framebuffer[i,j] ← s
        end if
    end for
end for
```

This is done polygon by polygon to determine the pixel color - done in the fragment shader. To implement this in openGL the following will need to be done:

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
glEnable(GL_DEPTH_TEST)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

13.2.2 Painters algorithm

Painters algorithm works in the object space: it draws objects further away first, then draws objects closer and closer on top of these further objects - it is good for transparency intteractions. This algorithm is not as efficient as the z buffers algorithm - it mimicks how a painter would deal with this issue.

This algorithm essentially works out which to paint first (furthest away) and paints the next layer at a time until all layers are completed THis algorithm fails when there are intersections or alternating overlaps:

13.3 Double buffering

In animation we will need to update the buffer, a swap buffer will do the following:

1. Make animation
2. Draw, and then repeat

Every time we make a change we post for a redisplay, then the currently displayed buffer will be swapped by the latest modified buffer, in this double buffering method.

```
glutInitDisplayMode(GLUT_DOUBLE)

void mydisplay()
{
    glClear(......);
    glDrawArrays(...);
    glutSwapBuffers();
}
```

13.4 Idle function

The idle function is a good place to store the animation and redisplay as it will only be executed once per redisplay loop.

13.5 Element buffers

We can reuse a vertex edge for multiple triangles in a single polygon. THis is done to save GPU memory and rendering time

Normal: Vector that goes directly out of a face

14 Lecture 14

14.1 Computer viewing

In implementing a computer viewing pipeline there are 3 main key considerations:

- Positioning of the camera (setting up the model-view matrix)
- Selecting a lens (setting up a projection matrix)
- Clipping (setting the view volume)

In openGL initially the object and camera frames are the same:

- Camera is located at origin and the camera is in the negative z direction
- Default projection is orthoganal
- Default clipping volume is a cube of size 2

14.1.1 Moving the camera in the frame

A move of a camera is a translation of the model-view matrix. The following will cause thise translation:

```
Translate(0.0,0.0,-d)
```

The camera can be moved into any position by doing a series of model-view translations and rotations. Remember that:

$$Matrix = Translations \times Rotations$$

Note: This order is commutative and as such matters.

```
// Using mat.h
mat4 t = Translate (0.0, 0.0, -d);
mat4 ry = RotateY(90.0);
mat4 m = t*ry;
```

14.1.2 Projection

Perspective: objects close to us are larger than objects further away

Default projection in OpenGL is the orthagonal projection (parallel) as apposed to perspective projection.

`glOrtho()`: is used for orthographic projections

In an orthographic view the focal distance is infinite.

14.1.3 Simple perspective

`d`: is the focal distance in openGL

This can be implemented in the follwing way by using:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}, p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\implies q = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

Frustrum: Provides the perspective projection

The frustrum can often be defficult to use, therefore will often wish to use the perspective function.

```
mat4 Frustum(left,right,bottom,top,near,far)
```

15 Lecture 15

Shading is a way in which we are able to determine the 3 dimensions of an object - different parts of the same object are shaded differently to discern depth.

15.1 Shading

Objects and lighting interating to give the appearance of the object on the screen. Flat shading gives the appearance of only 2 dimensions. Height light interactions cause each point of the object to take on a seperate color.

To create this shading we will need to consider:

- light sources
- Material properties
- Location of the viewer
- Surface orientation

Scattering: Some scattering light from object A will hit object B that may be scattered back onto object A again. Some of this light will eventually be scattered back to the camera for viewing.

Rendering equation: Describes the way in which light interacts with objects and the viewer in computer images. This equation is unable to be solved but has some decent approximations that are used in image rendering.

No shadows will be generated when the lighting calculations are done in the vertex shader as is vertex is dealt with on its own, no other vertex is therefore able to cast shadows.

In OpenGL the light is seen by all objets in the scene.

15.2 Local vs Global rendering

To correctly implement physical shading calculations we would need to implement **global lighting calculation** involving all objects and light sources interacting. This is simply not possible with current systems and we must instead take a "good enough" **local lighting calculation** - this emans shadows are difficult to implement.

15.3 Light material interactions

Light hitting an object is partially obsorbed and partially scattered. The amount of light scattered will determine the brightness and color of the object. The scattering of the light is determined by the smoothness and orientation of the surafce. The mots important factors in light material interactions are:

- Color
- Orientation
- Smoothness

15.3.1 Light sources

In the real world a lighht is not a point source - in openGL the light is a point source.

A point light source will have:

- Position: $(x, y, z, 1)^T$
- Intensity: (I_r, I_g, I_b)
- Illumination equally in all directions
- Can be affected by some attenuation factor (distance calculations)

A spotlight:

- defined by some cone
- A direction
- An intensity
- A position

A directional light source is:

- Parallel

Ambient light:

- There is some light everywhere in the scene
- irrespective of direction
- Will have RGB components

These different light types can be used in conjunction to make a scene

15.4 Phong model

Uses **diffuse**, **Specular** and **ambient** terms to compute the lighting. It is simple and can be computer raidly using the four vectors:

- $\vec{l}$ - The light source
- $\vec{v}$ - to camera
- $\vec{n}$ - to normal
- $\vec{r}$ - the perfect reflector of $\vec{l}$ with respect to $\vec{r}$

15.5 Surface types

Difuse: Specifies how rough a surface is

Lambertian surface: Perfectly diffuses light in all directions, and the reflected light is proportional to the vertical comopnent of the incomming light.

Specular: How smooth a surface is due to the incoming light being reflected in all directions

Specular reflections are modelled using:

$$I_{ref} \propto k_s I cos^{\alpha} \phi$$

Where α is some shininess coefficient. The bigger the shininess coefficient is, the shinier it is: $100 \leq \alpha \leq 200$ is for metal, $5 \leq \alpha \leq 10$ is for plastic.

Ambient light: For interatctions between large sources and other objects in the scene.

16 Lecture 16

16.1 Distance

Distance effects the intensity of light on an object. This relationship is an inverse relationshop. The calculations fo distant light sources is the same to parallel light sources (position does not matter).

Point source: $(x, y, z, 1.0)$ Distant source: $(x, y, z, 0.0)$ as the 0.0 is infinite

To calculate the attenuation factor the following equation is used:

$$\frac{1}{a + bd + cd^2}$$

where d is the distance from the light source, and a, b, c are some constant values to mimick the point source having some kind of size. This attentuation factor is only applied to the specular and diffuse terms, as the ambient light is there to emulate object-object scattering, and as such the distance does not play an as big role.

16.2 Blinn reflection model

While the phone model offers a decent fast lighting calculation system, it is often not very pretty (as it works on a per vertex basis). The phong model can be summerised as:

$$I = k_d L_d(\vec{l}\vec{n}) + k_s L_s(\vec{v}\vec{r}) + k_a L_a$$

Soecykar if the phong model is probeblatic as it requires a recalculation of the reflection term for each vertex. Blinn suggested we just simply use an approximation for this $\vec{r}$ value using the **halfway vector**. These changes are now modefied in the following equation

$$I = k_d L_d(\vec{l}\vec{n}) + k_s L_s(\vec{n}\vec{h}) + k_a L_a$$

This model is the default lighting calculation model in openGL.

16.2.1 Normal vector

At every point the normal vector will need to be found. This vector can be found b using the 3 points of the triange: cross product for 2 sides. OpenGL leaves the determination of the normal vectors to the application program:

$$n = (p_2 - p_0) \times (p_1 - p_0)$$

17 Lecture 17

To calculate the shading we will need to know:

- Normal vector of a vertex
- material properties
- Light vectors

GLSL has a built in normalization function to normalize vectors to unit length. We will normally divide vectors by their normal vectors to get the unit vector:

17.1 OpenGL light sources

```
vec4 light0_pos = vec4(1.0, 2.0, 3.0, 1.0);
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);
```

OpenGL does not have attenuation, and must be added by the application program. Material propertise of the object will need to have the same properties as the light:

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(0.8, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0;
```

Note: This does not change the relative RGB values as this would change the color - it is all done in proportion.

Transparency is specified by the alpha value (final value in the vector).

17.2 Polygon shading methods

17.2.1 Flat shading

Handles a polygon as having 1 normal giving the appearance f being a 2 dimensional object, or with many discrete faces

Advantages: Computationally cheap

Disadvantages: Edges come up when rendered

This type of shading is evoked by the following:

```
glShadeModel(GL_FLAT)
```

17.2.2 Smooth Shading

Gourard shading The normal of each vector is computer. The light level is next found by using the vectors $\vec{l}, \vec{n}, \vec{v}$. Shading points interior to the polygon are found by interpolating. This is described as **per vertex shading**.

Advantages: Gives much better looking results

Disadvantages: More computationally intensive than flat shading

Phong Shading Instead of using the vertex intensities to calculate the interior of the polygon, we use the interpolated normals at each fragment to compute the intensities of a fragment. This is described as **per fragment shading**.

Advantages: Very good physical lighting model

Disadvantages: Very computationally expensive compared to smooth shading.

18 Lecture 18

As a means of avoiding the ened to calculate millions of triangles on the GPU, we will use texturing, as apposed to subdivision. THere are three main types of texture mapping:

Texture mapping: Pasting an image (real or rendered) on top of the model to give it the appearance of texture

Bump Mapping: To change the local shape we will often use some form of bump mapping (small variations in the normals).

Enviroment mapping: Allows us to use an image of the enviroment as essentially a light.

To save computational power the texture mapping will occur right at the end of the render pipeline - in the fragment shader. To do these texture maps the following coordinates will need to be used:

- Parametric coordinates
- Texture coordinates
- Object/World coordinates
- Window / Scene Coordinates

Backwards mapping: Mapping from screen coordinates to texture coordinates

Given a point on the screen, backwards mapping will find the object it comes from and the corresponding texture.

The problem is not just mapping points to points, but also areas to areas - a square of the 3D surface mapped to the area of a texture.

Aliasing: The process by which a smooth curve or line becomes jaggered due to the resolution of the graphics device or frmo insufficient data to represent the curves

Aliasing can come about by taking periodic samples of points on a texture - Will generally occur when zoomed out, as it is the averaging of the texture that will be seen.

18.1 Two part mapping

1. Look for an intermiadiate object that is closest to the 3D object we wish to map the texture onto
2. Map the texture onto the actual object

18.1.1 Part 1

An intermediate object will need to be easility parametized, suchas a sphere, cylinder or box - This is why spheres are often used in enviroment maps. As there will be distortion we must decide where to put this distortion.

18.1.2 Part 2

There are 3 ways to map the texture to the real object from the intermediate object:

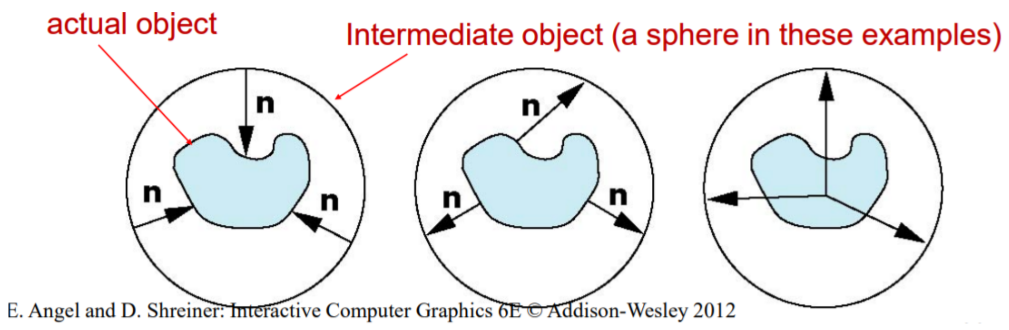
1. Normals from the intermediate to the actual
2. Normals from the actual to the intermediate
3. Vectors from the center of the intermediate

19 Lecture 19

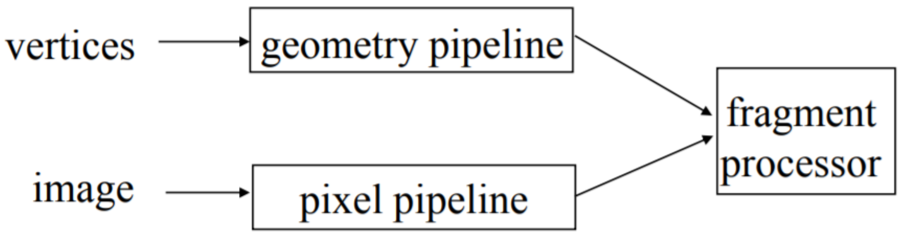
There are three steps to applying textures:

1. Specify the texture

    - read or generate images
    - Assign to texture
    - Enable texturing


actual object
Intermediate object (a sphere in these examples)
n
n
n
n
n
n
E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Mapping textures from an intermediate object to the actual object



Geometric and Texture pipelining

2. Assign vector coordinates to the vertices
   - Proper mapping function is left to the application
3. Specifying vector parameters
   - Wrapping or filtering

19.1 Texture mapping in the OpenGL pipeline

As images and geometry flow through sepearate pipelines, complete textures have no geometric effect on complexity.

The geometric pipeline does far more heavy lifting therefore it is preferable to have the complexity in the image pipeline, and then have them mergered at the end of the pipeline in the fragment shader. In OpenGL this is done in the following:

```
GLubyte my_texels[512][512];

glEnable(GL_TEXTURE_2D)

glTexImage2D(target, level, components, w, h, border, format, type, texels );
```

An example of this is in the following:

```
// pass the vertex coordinates to vertex shader
offset = 0;
GLuint vPosition = glGetAttribLocation(program,"vPosition");

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE,0, BUFFER_OFFSET(offset) );

// piggy-back the texture coordinates at the
// end of the buffer and pass it to vertex shader
offset += sizeof(points);

GLuint vTexCoord = glGetAttribLocation(program,"vTexCoord");

glEnableVertexAttribArray( vTexCoord );

glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );
```

19.2 Interpolating

When a shape is more complex than a rectangle there will become distortion in others to form the texture around the object. The following values will need to be set to optimize the outcome of the distortion:

- Wrapping: Paramater determines what happens if s or t are outside of the range
- Texture sampling: Specifies average area rather than a point
- MipMapping: allows use of textures at multiple resolutions
- Enviroment parameters: Determines how the texture mapping interacts with the shaders

When imported the texture will always be converted to a value between 0 and 1.
These parameters can be set by:

```
glTexParameter*(GLenum target, GLenum pname, GLint value);// * can be i or f
```

19.2.1 Wrapping

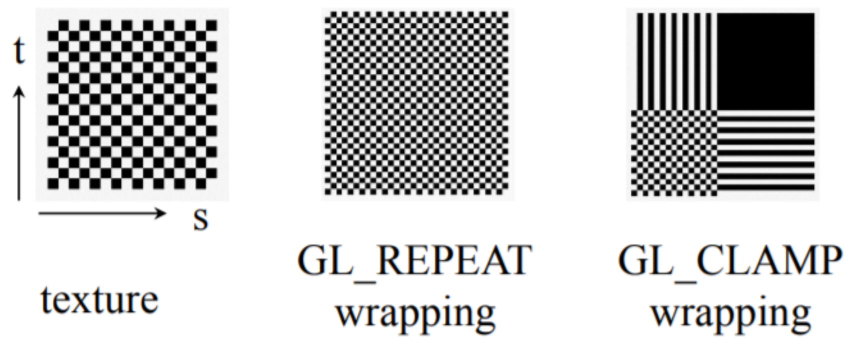
19.2.2 Texture sampling

Point sampling: Use the texel which is closest to the texture outputs of the rasterizer
Linear filtering: A weighted averaage of the group of texels in a neighborhood of the texture coordinates.

19.2.3 Magnification and minification

Magnification: The texel is larger than one pixel
Minification: The texel is smaller than one pixel

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```



Example of wrapping heuristics

If an object is always being minified we can generate a minimised texture - this is referred to as **Mipmapping**.

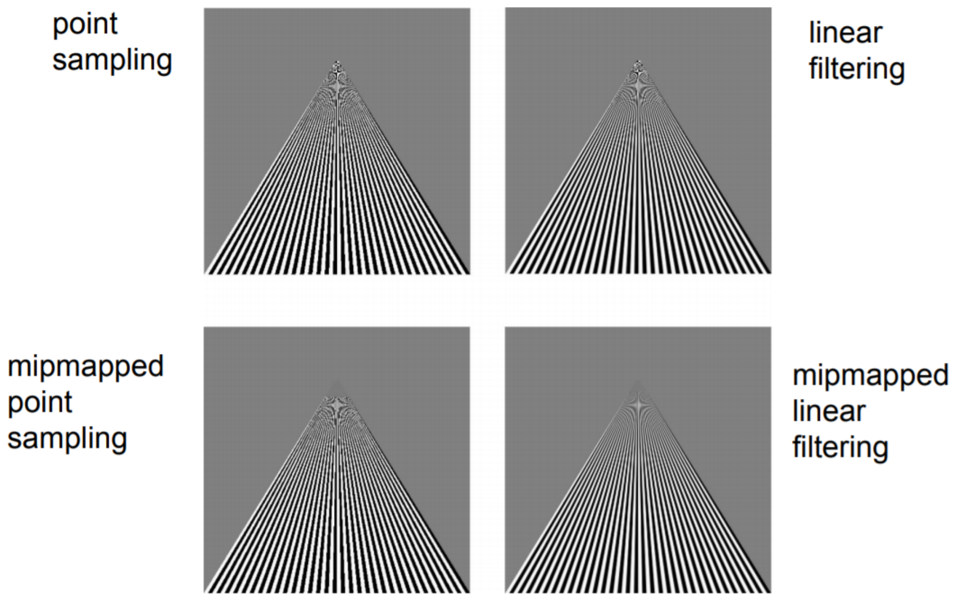
MipMapping An efficient way of storing multiple copies of different resolutions of the same texture to avoid interpolation.

```
glTexImage2D( GL_TEXTURE_2D, 0, ... );
glGenerateMipmap(GL_TEXTURE_2D);
```

19.2.4 Using texture objects

1. Specify texture objects
2. Set texture filtering
3. Set texture function
4. Set texture wrap mode
5. Set optional perspective correction hint
6. Bind texture object
7. Enable texture
8. Supply texture coordinate for vertex

```
in vec4 color; //color from rasterizer
in vec2 texCoord; //texture coordinate from rasterizer


uniform sampler2D texture; //texture object from application

void main() {
    gl_FragColor = color * texture2D( texture, texCoord );
}
```



Example of effects of different texturing techniques

20 Lecture 20

A complex object can often be decomposed into simpler primative parts as symbols. TO render these complex objects we can then draw further complex objects, defined as an instance transformation. This can be represented in one transformation in the form:

$$M = TRS$$

Symbol instance tables: Contain the M for each instance of the symbols

A tree or graph model can be used to give a parent-child relationship between instances of objects

20.1 DAG model

A directed acyclical graph will be formed due to the fact that different objects will have matching symbols - more efficient than a tree model.

When using a tree model we must find the relationship between nodes to give the dependencies:

- The nodes can store teh object
- The edgest can store the transformations

20.2 Articulated models

A robot arm is an example of an articulated model which:

- Connected parts are connected to a join
- We can specify the position using relative joint angles

A generalization of tree traversals will be required in order to deal with larger cases. For the graph traversal we will need to visit each node to calculate the transformations. Depth first search is perfect for this sittauion as it allows for association with a parent-child relationship without multiple traversals (ie. Do translation of parent, then use the parents translation to also make a child translation).

Skinning: Weighted average of transformations to create more realistic joint connections.

21 Lecture 21

21.1 3D modelling

Models can be sourced using:

- Scanners
- Non rigid deformations of linear models
- Computer generated models

Typical 3D models will sue high level controls to avoid the tedious nature of changing individual vertexes.

21.1.1 Subdivision

Subdivision: Producing smooth surfaces that can be adjusted easily

The idea is to specify a blocky surface with a managable number of faces and then let the program (such as blender) convert it into a smooth vertex object. This smoothing system needs to be predictable.

Catmull-clark Catmull-clark subdivision surfaces technique is used to turn the control surfaces into smooth surfaces as it is: Simple, predictable and features:

- Each original point effects a small part of the surface
- The first derivative is always continuous
- The second derivative is almost always continous

This is the surface subdivision system used in blender. The rate of change of the normal is continuous, therefore the surface is smooth.

In subdivision situations we want to work with the minimum number of faces possible to maintain the desired amount of detail.

The algorithm is as follows:

After each subdivision there is a new vertex added at each:

- Old vertex
- Old suface
- Old edge

To determine where this new point will be placed the following equation is used:

$$\frac{F + 2E + (n-3)P}{n}$$

Where:
F is the average face points E is the average edge points n is the number of edges
The faces and the edges are the origional ones that touch the original P.

- This can be used to find texture
- Unlike nurbs, catmull clark can be extended to other topologies
- More predictable for open surfaces

The standardised form of the number of vertexes, edges and faces can be found in the following:

$$V_n = V_{n-1} + F_{n-1} + E_{n-1}$$
$$F_n = 4F_{n-1}$$
$$E_n = 2E_{n-1} + 4F_{n-1}$$

As the number of subdivisions increases of a cube, we approach a perfect sphere.

22 Lecture 22

22.1 Animations

The main idea of animations are keyframes

Keyframes: A key frame is a signification point of action , the rest of the frames are interpolated

The position between the frames must be interpolated, normally using some kind of linear interpolation. Note that keyframes do not need to be equidistant apart - they can occur at any period, and any, varying, frequency.

22.1.1 Problems with interpolation

- unexpected results can occur when we combine three interpretations
- Gimball lock can occur when 2 degrees of freedom align with eachother

Using eulers angles, gimball lock is extremely difficult to avoid

22.2 Quaternions

These are an extension of complex numbers $ i, j, k$, thet satisfy:

$$i^2 = j^2 = k^2 = ijk = -1$$

with real numbers:

$$a, b, c, d \in \mathbb{R} \implies q = (a, b, c, d)$$

which are computed with

$$a + bi + cj + ak$$

The following cases can occur:

- When $a = 0$: **Pure Quarternion**
- When $||q|| = 1$: **Unit quarternion**

There are a few advantages of using quarternions, over using eulers angles:

- Interpolation is easy between 2 quarternions
- Smoother animation can be achieved
- Quarternions take less storage spage
- Easily converted to matrices for rendering
- Do not suffer from gimball lock

These will still need to use standard translations and scale, saving no space in practice - however the advantages outweight those of eulers. These will need to be implemented for use in OpenGL.

23 Lecture 23

23.1 Model meshes and bones

Skins: The meshes of a 3D object **Bones**: The articulated underlying components of a model for animations

For natural animations we wish to have more fluid animations than ridgid transformations.

Rigging: Is used to create a low level tranformation that allows for controls to be connected to a model.

Skinning: Provides a means to create animations while allowing for minimal folds and creases at the joints.

All animatomical models will be made up of the skeleton and a skin. This system can then be used for articulated, more realistic, animations - without the need for constant re-modelling of the mesh per frame or keyframe.

23.1.1 Implementation

1. Define bone heirachy in some form of data structure
2. Note that vertices can be affected by multiple bones
3. Bones will have a weighted affect on the vertices - how much is it affected by each bone

The graphics software will be used to assign weights for each vertex by the bones:

- Automate weighting
- Weight painting
- Defineng envolopes

To animate: once the bone structure has been rigged to the skin, then the bone structure can be rotated to create natural movements - the mesh will follow the bones

RestPose: All animations are defined in terms of some rest pose

23.1.2 Bone movement

A 4×4 matrix is modified per bone, they are defined in terms of some bone weights. Bones will transform:

- Children bones
- Corresponding vertices
- The normal vertex

And then the weighted calculations will be done to determine where the corresponding vertex will be due to the weight of the bones.

Note: the normal will be tranformed with the vertex to save some computation time.

The sum of all the weights of the vertex should sum up to 1 (ie. 100%).

23.1.3 Skinning algorithms

1. Tranform vertex positions from the object space from the rest pose are tranformed into bone space using **InvBMs**.
2. Performs the keyframe animations for that frame which transforms the vertex position back into skin space in its animated pose
3. The model is then rendered

GPU Skinning Smooth skinning will be done on a per vertex basis, as it is ideally done for every vertex - This means that it will be done within the vertex shader of the processessing pipeline.

To generate each of these calcualations:

- Using "varying" attributes to the vertex shader
- These dont change once the model is loaded
- Normally only a small number of bones will affect each vertex to some computational time.

These varying variables will be changed each time the model is sent to the vertex shader - ie. Every time an animation frame occurs. These bone weight relationships per vertex is generally static - why would the affect of a bone on vertexes change over time in most animations?

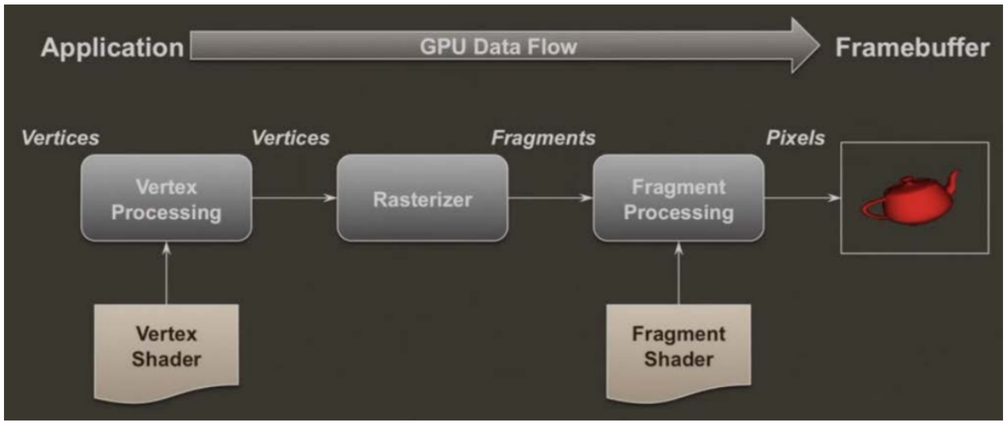
DirectX files: Contains all the matrix transforms for animations, and the bone structure and skinning.

24 Lecture 24

WebGL is an implemetations of OpenGL that uses javascript and the HTML canvas element to display graphics similar to those in the standard openGL library, in the browser.

All webGL programs will need to:

- Set up a html canvas to render into
- Generate data in applications
- create shader programs
- create a buffer and load data into them
- "connect" data locations with shader variables
- render



The rendering pipeline

The webGL application will consist of:

- A htmls file for structure
  - Individual shaders
  - Html reads in utilities and applications
- A js file (or embedded) for application logic.

24.1 WebGL pipeline

24.1.1 Vertex shader

- Shader is executed for each vertex
- Output is passed to a rasterizer
- Positions outputs in clipped coordinates
- Carrys out effects:
  - Changing coordinate systems
  - Moving vertices
  - per-vertex lighting

24.1.2 Fragment shader

- Shader is executed for each potential pixel
- Will need to pass serveral tests before making it to the frame buffer
- Many effects ca be done in the fragment shader:
  - Per-fragment lighting
  - Texture and bump mapping
  - Enviromental maps

24.2 Application breakdown

24.2.1 HTML

- Contains shaders
- Brings in utility and application JS files
- Describes page elements: buttons, menus, etc.
- Contains canvas element

24.2.2 JS

- `init()`
  - Sets up **VBOs**
  - Contains **listeners** (callbacks) for interactions
  - Sets up required transformation matrices
  - Reads compiles and links shaders
- `render()`

Vertex data is used to make up the shapes and is expressed in terms of 4D homogenous coordinates and this vertex data is stored in the **vertex buffer objects** (VBOs). As arrays are fundamentally different in java script to those found in C or C++, we will need to use the `flatten` function to normalize them:

```
flatten(array)
```

Which can be found in the example:

```
gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);
```

24.3 Drawing geometric primatives

- Before drawing, frame and depth buffers must be cleared
- The depth buffer is used for hidden surface removal
- `glDrawArray` initiates the vertex shader for the drawing
- `requestAnimationFrame` is needed for redrawing if anything is changing