Introduction to OpenGL

Andreas Danner Nilsen
andreasd@pvv.ntnu.no

Plans for today

- OpenGL as an API
  - History, development
  - Advantages and Disadvantages
  - Howto get started

- Theory and examples
  - Simple Drawing
  - Vertex transformation pipeline
  - Pixel Testing
  - In-depth Texturing
  - Lighting crashcourse
  - Vertex Buffer Objects
  - Sorry, no time for shaders...

What is OpenGL?

- IrisGL from SGI, competing vs PHIGS
- 1992: OpenGL 1.0 released (ARB)

- First 'open' 3D api for common use.
  - Hardware independent
  - Widespread use in university and CAD circles

- OpenGL is a *rasterizer* API
  - Transform 3D geometry to 2D images

What is OpenGL?

- OpenGL doesn't know about OS, windowing libraries or anything beyond rasterization
  - Very dependent on the window API GL bindings
    - GLX (unix)
    - WGL (windows)
    - AGL (apple)
- OpenGL decides the content of a surface
  - But not anything else

OpenGL today

- Version 2.0: Shaders
- Khronos: OpenGL ES
- ARB / Khronos
  - Long Peaks
- Version 3.0: well.. uh...
  - Deprecation model
- What now?

Where to start

- Often the hardest problem – issues!
- Windows: WGL stuck at version 1.1
  - wglGetProcAddress / GLEW
- Linux: Restricted drivers, glu hell
  - Gotten a lot better since my last attempt ;)
- Mac: AGL/GLX interaction issues
  - Still error prone

OpenGL issues

- Retaining OS independency
  - GLUT
  - SDL
  - EGL
  - Homebrew solution
- Direct X replacements?
  - SDL is not enough
  - EGL definitively not enough
- OS independency – hard.

What I did...

- For today I'll be using a homebrew solution

- dglCreateWindow
- dglDestroyWindow
- dglSwapBuffers

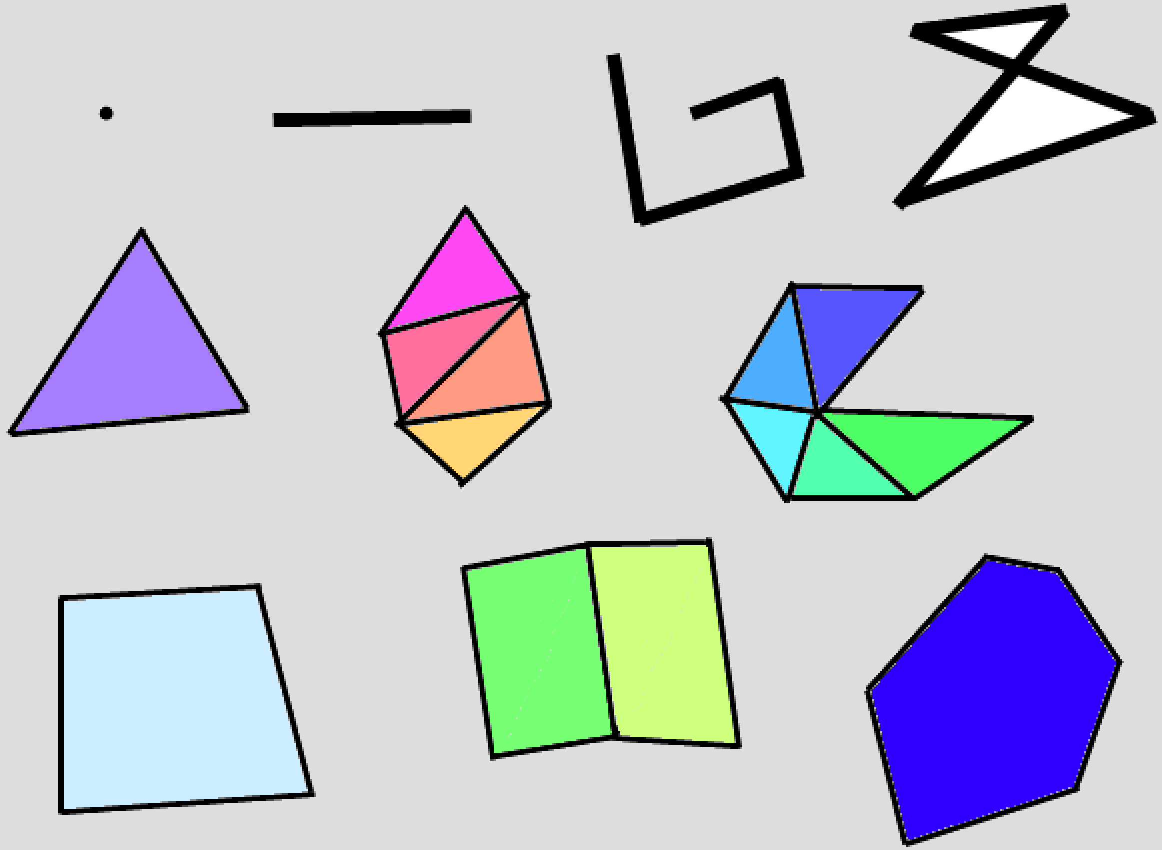
- Lots of fun to make your own wrapper library
  - Takes a lot of time

Simple Drawing

- OpenGL is a rasterizer.
- Converts primitives to 2D images

- Primitives:
  - Points
  - Lines
  - Polygons

- Provide GL with primitives and *that's it*



Vertices

- 'Edge points' for primitives
  - 2 for lines, 3 for triangles, 4 for quads
- Each vertex have a position
  - Given as an affine value
    - x, y, z, w
- Think of w as a divisor
  - Real x = x / w
  - Real y = y / w
  - Real z = z / w
  - 'If w = 1, it can be ignored'

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

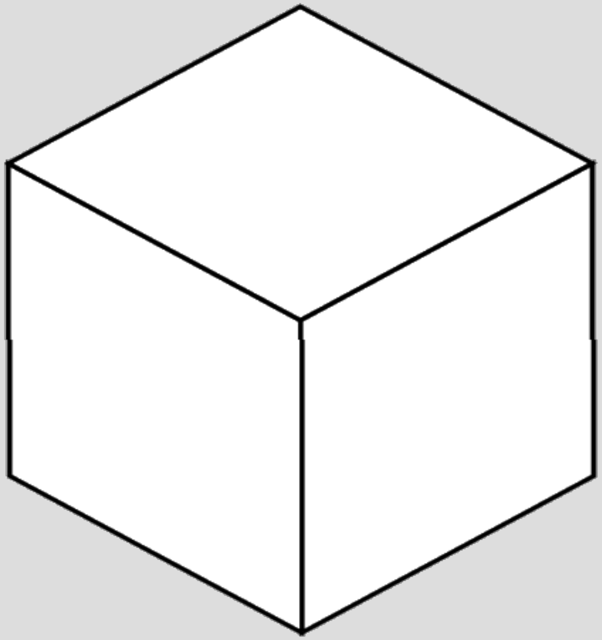
Clipspace – 'OpenGL world'

- Origo is the center of this cube
- Camera looking at origo from along the z axis
- Top, bottom, left and right walls limits the screen

- What are the two last walls?
  - Nearplane
  - Farplane

- All walls at -1, +1

- Anything outside this cube is *clipped*.



Drawing a Quad

- Consists of 4 vertices
  - Each vertex has a position
- OpenGL likes geometry in CCW order
  - Can be changed, but let's play nice

Vertex1 = <	-0.8,	-0.8,	0.0	>
Vertex2 = <	0,8,	-0.8,	0.0	>
Vertex3 = <	0.8,	0.8,	0.0	>
Vertex4 = <	-0.8,	0.8,	0.0	>

- Need to pass this data to OpenGL

glVertexPointer

- Accepts an array of vertex positional data
- Takes four parameters
  - size - 2, 3 or 4. Padded with [0,0,0,1]
  - type - usually GL_FLOAT
  - stride - distance between vertices, or 0
  - pointer - a pointer to the data

- Allows GL to extract positional data from almost any memory construct.

- Last but not least:
  - glEnableClientState(GL_VERTEX_POINTER);

glDrawArrays

- Draws stuff from the arrays given
  - Positional data retrieved from the glVertexPointer call
  - There are other arrays too!
- Takes three parameters
  - mode - what to draw, GL_QUADS for now
  - first - the first index to draw
  - count - number of indices to draw.
- glDrawArrays(GL_QUADS, 0, 4);

Drawing a Quad

- Time for an example!

glDrawElements

- Same as draw-arrays, but indirect.
  - Re-using indices
- Need an array of indices

```
unsigned char indices[] = { 0, 1, 2, 2, 1, 3 };
```

- glDrawElements(GL_TRIANGLES,
  6,
  GL_UNSIGNED_BYTE,
  indices
  );

- Use as conservative indextype as possible!
- Let's see this in action :)

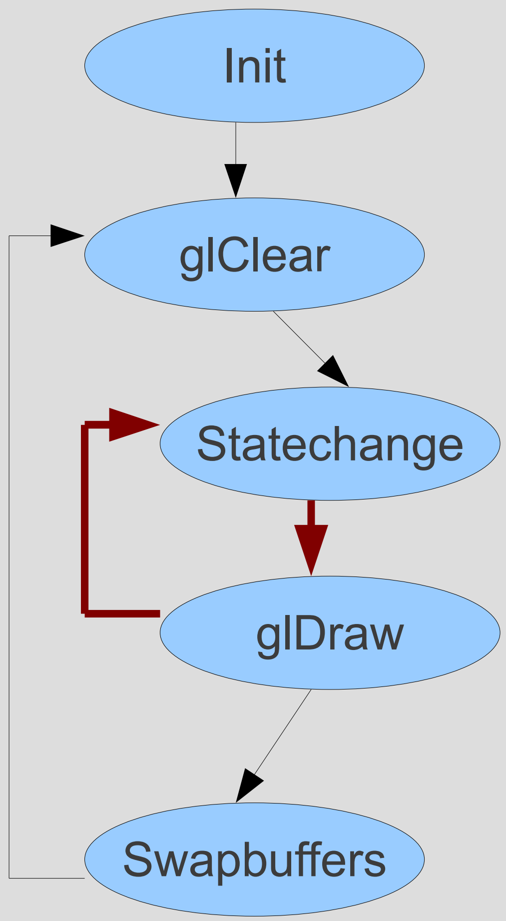
Colors are fun!

- Let's add another pointer
- glColorPointer
- Works just like glVertexPointer
  - Size, type, stride, pointer

- Again, remember to enable the pointer
  - glEnableClientState(GL_COLOR_POINTER);
  - Remember to disable this if not needed!

- Let's just do this with an example as well

Efficient use of OpenGL



- Statechanges are cheap
- Drawcalls are pipelinable

- Transition between draw and statechange is usually expensive (red arrows)
  - Varying with HW

- You *need* statechanges
  - Often possible to reduce

- *Scenegraphs* break this
  - But are usually worth it

glBegin/glEnd must DIE !!!!!

- All tutorials begin with these two
- They are outdated and SLOW
  - Tearing down program vertex arrays
  - Only to have the driver re-build them
  - Overkill of gl calls to draw anything
  - Unknown amount of attributes per vertex
  - Waste of internal driver allocations
  - Excessive amount of state set per vertex
  - Better to send the pointers instead
- Join the crusade today

Vertex Transformation Pipeline

- Placing vertices inside the clipspace cube is tedious!
- Use a good mathematical tool for this job:
  - Affine Transformations
- I'll skim through this fast
  - In depth on this next week!

Moving (Translation)

- Moving a vertex is easy
  - Simply add a value to the vertex component

- By adding the same value to *all* vertices
  - we can move everything.
- By adding the same value to all vertices in an object
  - we move the object

Scaling

- Scaling 'a vertex' is also easy
  - Multiply by some value per component
  - Looks kinda scary in maths

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} = \begin{bmatrix} x * S_x \\ y * S_y \\ z * S_z \end{bmatrix}$$

Scales around origo

- By multiplying the same value to *all* vertices
  - we can scale everything.
- By multiplying the same value to all vertices in an object
  - we scale the object

Affine transformations

- Combining these two

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} = \begin{bmatrix} x * S_x \\ y * S_y \\ z * S_z \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 & T_x \\ 0 & S_y & 0 & T_y \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x * x + T_x \\ S_y * y + T_y \\ S_z * z + T_z \\ w \end{bmatrix}$$

The colored 4x4 matrix is called an affine transform matrix
It holds both scaling and translations

Rotations

- Rotations are sort of like scaling
  - Rotates around an axis

$$Q_{\mathbf{x}}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix},$$

$$Q_{\mathbf{y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix},$$

$$Q_{\mathbf{z}}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

Rotates around origo

No, you can't rotate around all 3 at once.

Chaining Affine Transformations

- The point of affine transformations is that it can take ANY amount of transformations and squeeze them down to 16 numbers
  - Matrix Multiply the steps together
  - Order matters

Transformation = [Scale down by 0.75 in all three directions] * [Rotate by 35 degrees around y.] * [Translate by 16 units in the z axis] * [Rotate by 16 degrees around y]

- Then multiply the vertices by the matrix
- This matrix is called the *modelview* transform

OpenGL is easier!

- Builtin support for affine transformations

- glLoadIdentity - reset matrix to default
- glRotatef - axis to rotate around, and degrees
- glTranslatef - offset to translate in each axis
- glScalef - factor to scale in each axis

- To set up a modelview transformation matrix, simply call the GL calls in the proper order.

- OpenGL will apply the current modelview matrix on all vertices

Earlier example:

```
glScalef(0.75, 0,75, 0.75);
glRotatef(35, 0, 1, 0);
glTranslatef(0, 0, 16);
glRotatef(16, 0, 1, 0);
```

Example time!

- We really need an example for this one!

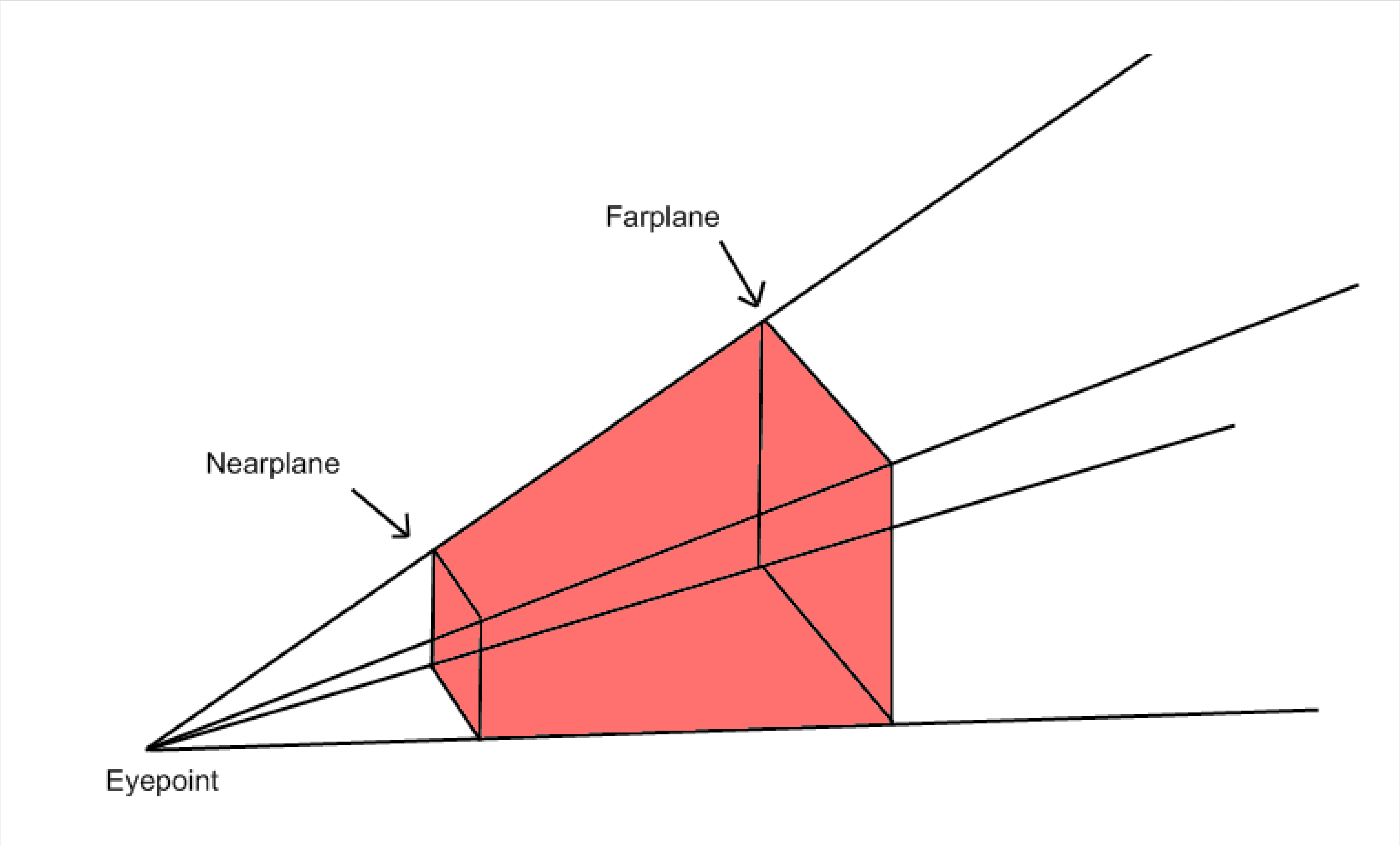
Camera!

- OpenGL has no concept of camera
  - Always looking at origo in clipspace
- Instead: Projection matrix
  - Kinda works like the modelview matrix
  - But mathematically applied before that

    *projection * modelview * vertex*

What is a Frustum?

- Decapitated Pyramid



Perspective

- Set up a frustum instead of a clipbox.
  - Works in the same way, only different shape
- Projection transform: from frustum to clipbox
  - Adding perspective 'resizing'

$$\begin{pmatrix} \frac{2\,near}{right-left} & 0 & A & 0 \\ 0 & \frac{2\,near}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$A = \frac{right+left}{right-left}$$

$$B = \frac{top+bottom}{top-bottom}$$

$$C = -\frac{far+near}{far-near}$$

$$D = -\frac{2\,far\,near}{far-near}$$

Setting up a Frustum matrix

- glFrustum
  - Left
  - Right
  - Bottom
  - Top
  - Near - Do not set to zero
  - Far - As far as you like ... but...

- Keep in mind, the eye is at origo

Setting the Matrices

- glMatrixMode(GL_PROJECTION);
- glLoadIdentity();
- glFrustum(-1, 1, -1, 1, 1, 500);
- glMatrixMode(GL_MODELVIEW);
- glLoadIdentity();
- glRotate(...); glTranslate(...); glScale(...);

Or even easier!

Thanks to GLU

- gluPerspective - setting up camera matrix
  - fov - field-of-view
  - aspectrate - width/height
  - nearplane - same as glFrustum
  - farplane - same as glFrustum

- gluLookAt - setting up modelview matrix
  - Eye - position of the eye
  - Center - the coordinate you look at
  - Up - direction up

- Example time!

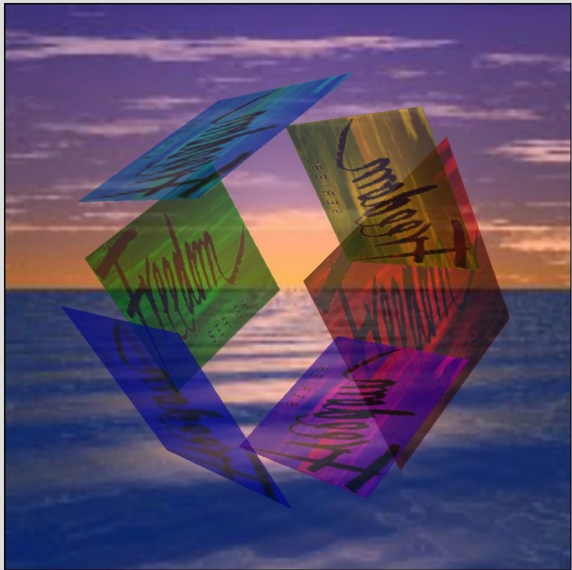
Advertisement

- Enough Matrices for now
- More on the subject next week, lykkebo

Blending

- Mix a draws pixel with the buffercolor
- glBlendFunc(sourcefactor, destfactor);
  - GL_ONE
  - GL_ZERO
  - GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA
  - GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA
  - And more
- glBlendFuncSeparate

- Order matter!



Result = source * sourcefactor + dest * destfactor

Pixel Testing

- OpenGL can be configured to NOT draw
  - Per pixel basis

- Depth Testing
- Alpha Testing
- Stencil Testing

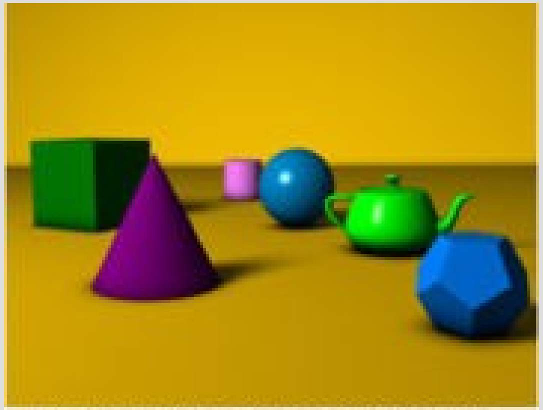
Depth Testing

- Painters Algorithm

- Buffer of z value per pixel
- Can configure to not draw pixels based on z value

- glEnable(GL_DEPTH_TEST);
- glDepthFunc(GL_LESS);

- Depthbuffer must be cleared per frame


A simple three dimensional scene


Z-buffer representation

Alpha Testing

- Drop pixels based on alpha value

- glEnable(GL_ALPHA_TEST);
- glAlphaFunc(GL_LESS, 0.3);

- Faster than blending

Stencil Testing

- Drop pixels based on custom per-pixel value

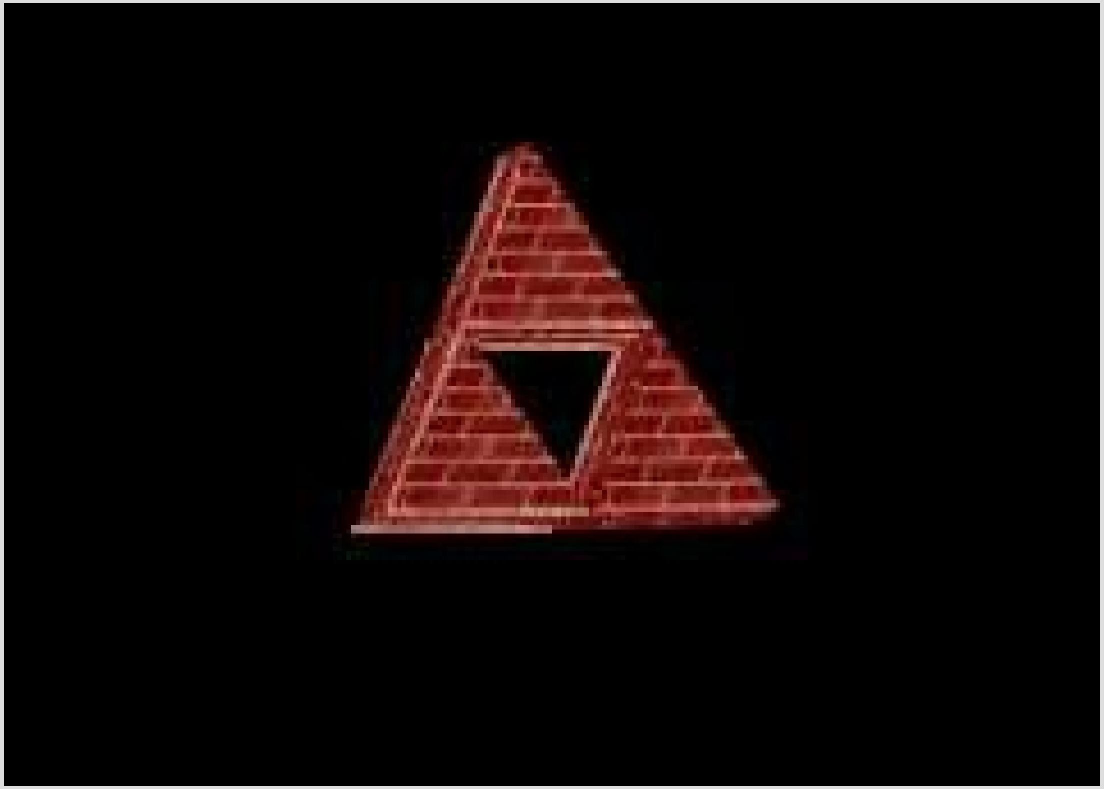
- glEnable(GL_STENCIL_TEST);
- glStencilFunc
- glStencilOp

- Useful for lots of stuff!
  - Stencilshadows, masking
  - Only creativity limits

- Stencilbuffer must be cleared per frame

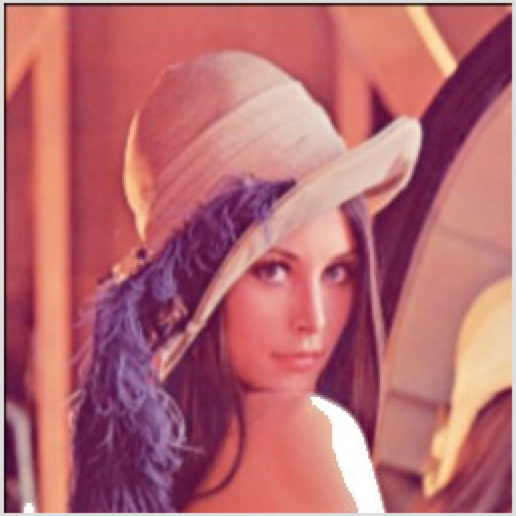
Texturing

- Adding images on top of your geometry



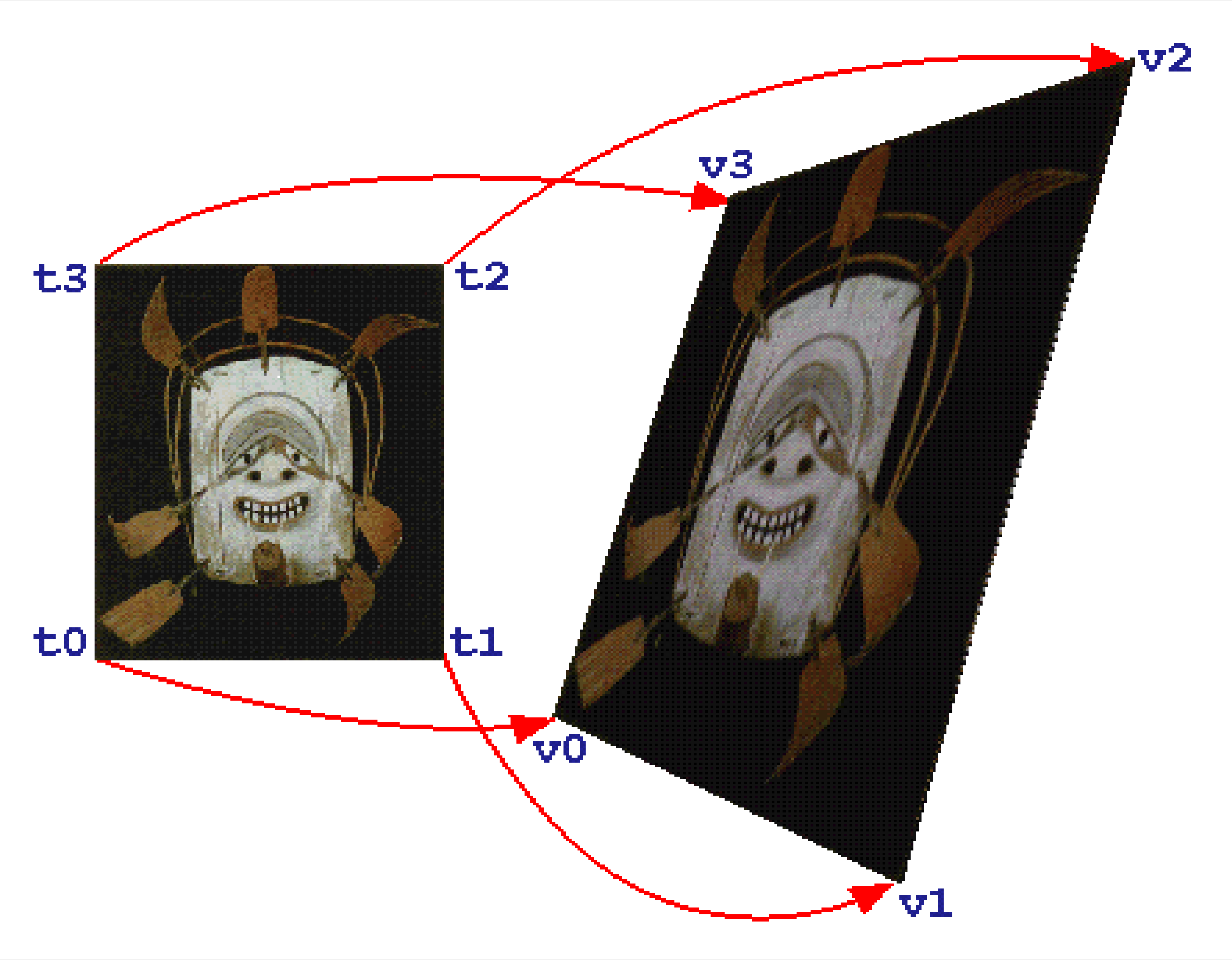
Texture Coordinates

- Like color, attribute per vertex
- glTexCoordPointer
  - Size - usually 2
  - Type, - GL_FLOAT or an integer
  - Stride, - like all other
  - Pointer



- Also needs enabling

Texture Mapping



OpenGL Object Model

- *Some* OpenGL state is wrapped in Objects
  - Textures
  - Framebuffers and Renderbuffers
  - Vertex Data Buffers
  - Shaders and Programs
- Objects can be *bound* to *targets*
  - Think of a target as a global variable
    - GL_TEXTURE_2D
    - GL_FRAMEBUFFER
    - Etc...
- Functions modifying objects work on targets, *not objects*

Creating Objects

- Objects are created when bound
  - glBindTexture(GL_TEXTURE_2D, someid);
- You *can* grab id numbers as you please
  - Bad idea, easy to mess up
- glGenTextures(arraysize, array);
  - glGenTextures(1, &some_variable);

Object Namespaces

- All objects are stored in different *lists*
- Each object has a 32bit ID number unique per list
- Object 0 often special, depending on type

Texture Object Properties

(mipmaps excluded)

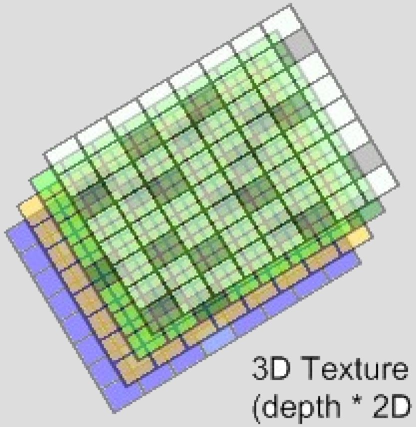
- Dimensionality -1D, 2D, 3D or Cube
- Width and height -Power of two?
- Data Format -RGB8, RGBA8, +++
- Wrapping rules -Clamp or Repeat
- Border -Usually 0
- Minification and Magnification Filter
- The texel data itself

Dimensionality

- One of these four

1D Texture

2D Texture



3D Texture
(depth * 2D Textures)

Cube Map
(6x2D Textures)

Texture Magnification Filter

- Two to choose from

GL_NEAREST (default)

GL_LINEAR
require 4x samples per pixel,
but this performance hit is usually
caught by the HW texture cache

Texture Wrap Modes

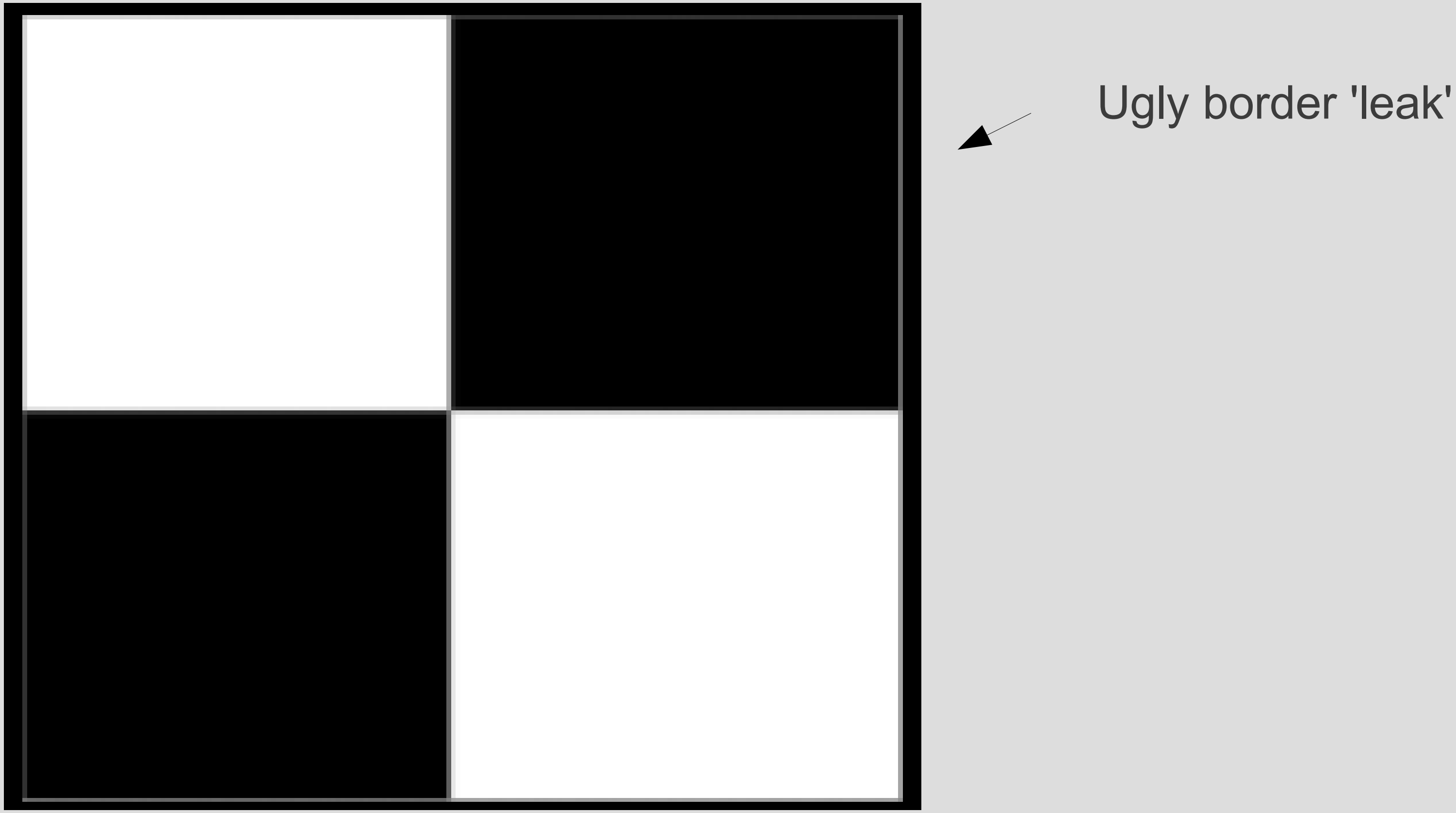
- Two to choose from

GL_REPEAT (default)

GL_CLAMP
GL_CLAMP_TO_EDGE

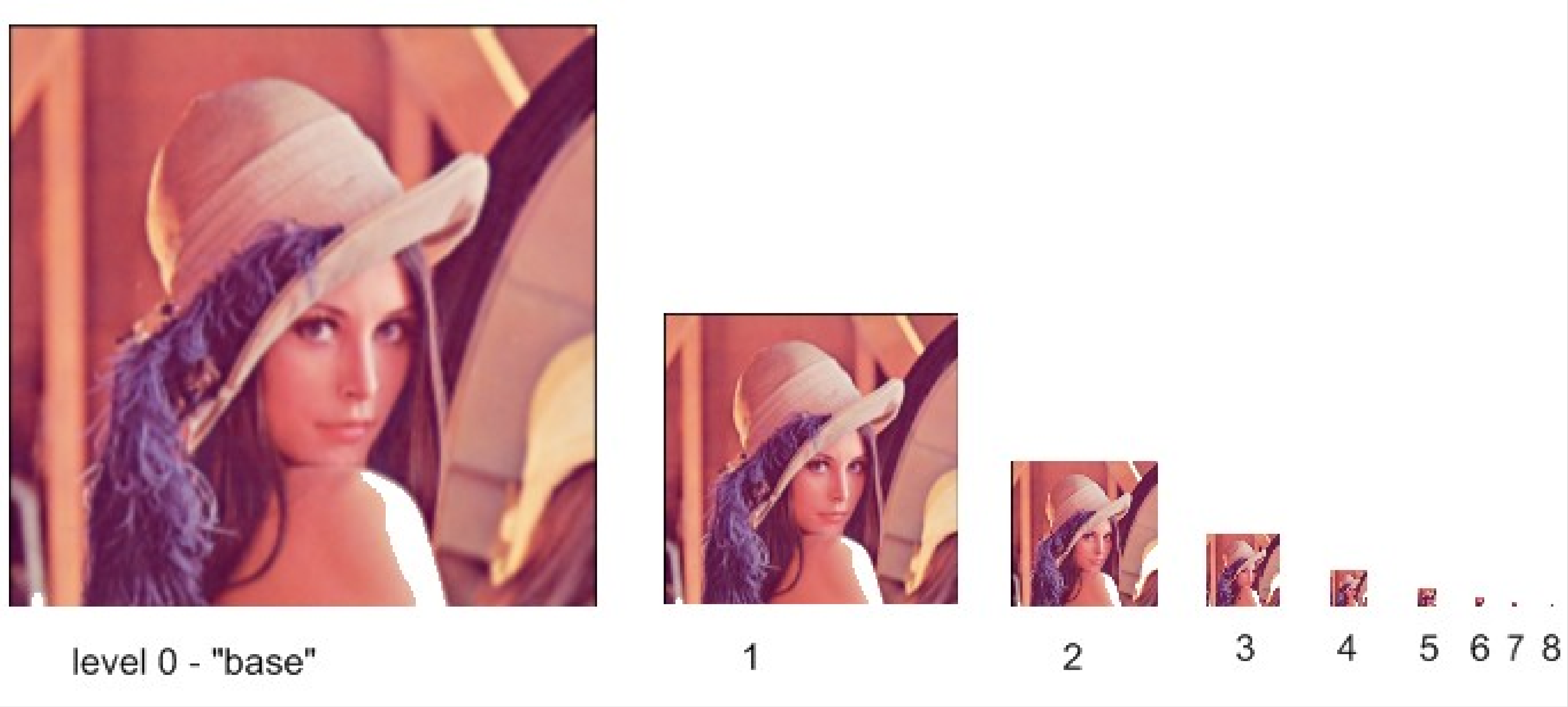
Texture Wrap mode - why?

- Magfilter Linear + Wrapmode Repeat leads to this
  - *May* be desirable for looping textures



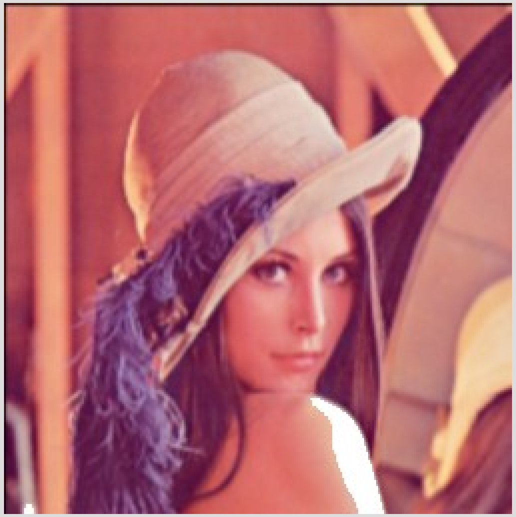
Mipmaps

- Smaller versions of textures



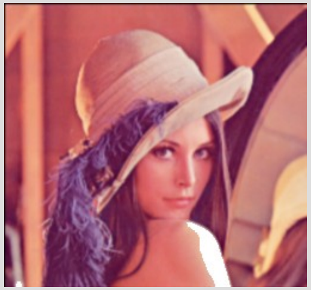
Miplevels

- Your GPU will pick the proper miplevel
- The one matching the size best
  - Or the two bounding miplevels...



Miplevel 4

Drawn Quad



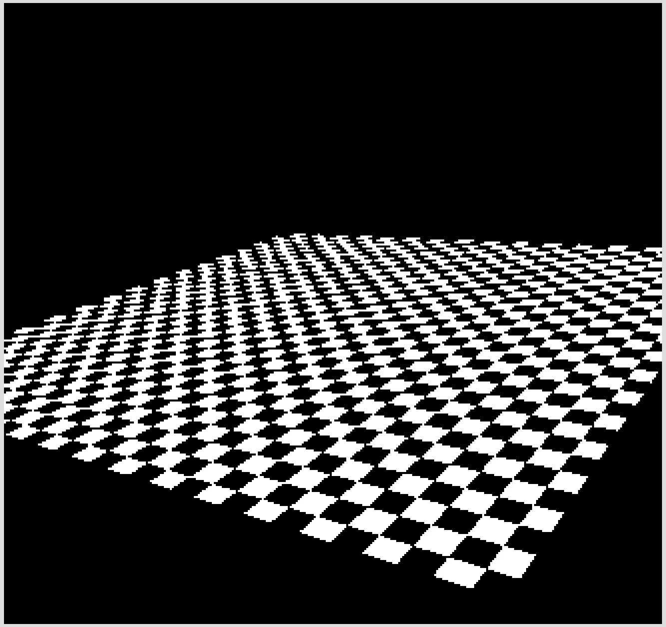
Miplevel 5

Mipmaps - why?

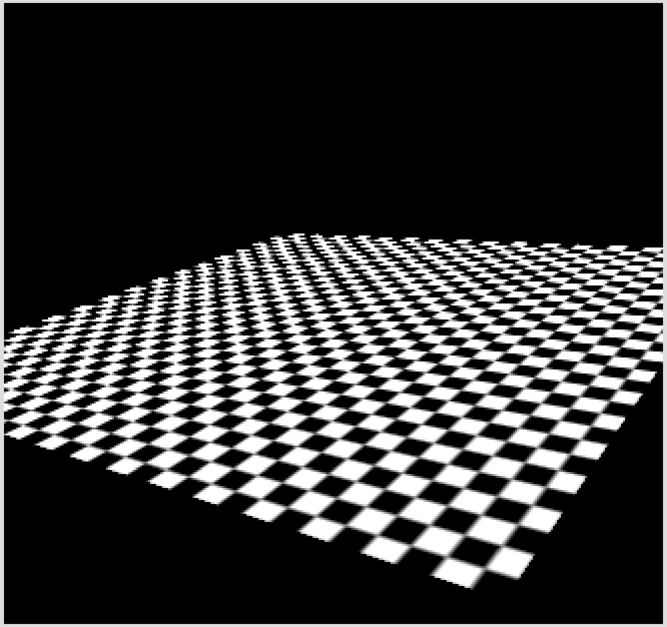
- Allows the GPU to sample in smaller textures
- Saves Texture bandwidth
  - Better speed
- Improved visual quality
  - The mipmaps *are* the best visible reduction
  - Better result than having the GPU do it
- Absolutely NO reason to not use mipmaps
  - Barring lazyness or 1:1 overlays
- You can specify all mipmaps yourself ... or ...
- OpenGL can generate mipmaps for you

Texture Minification Filters

- Without mipmaps, choose from these two



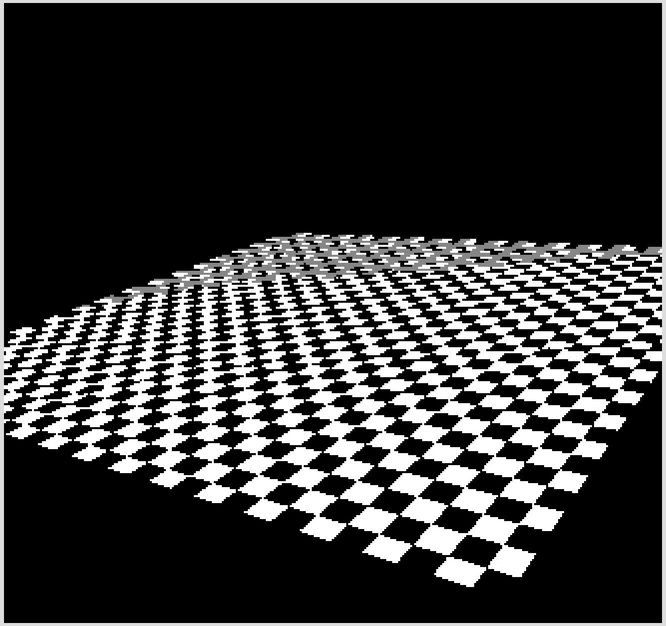
GL_NEAREST
same performance hit as
magnification filters



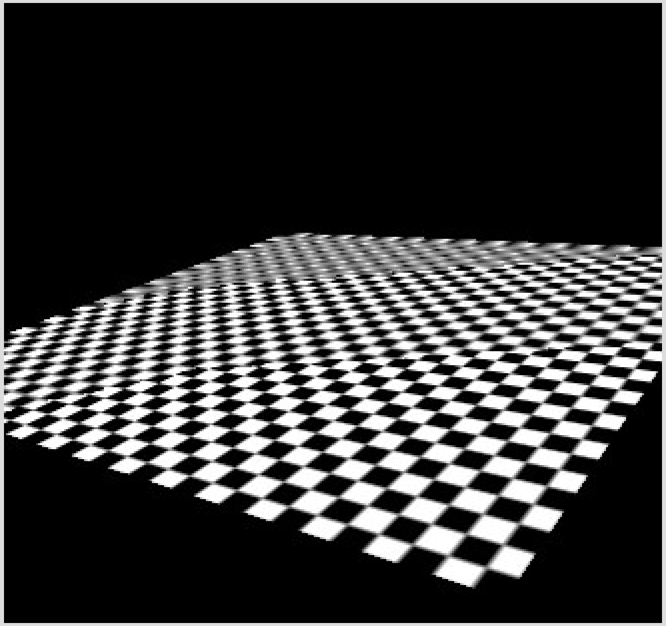
GL_LINEAR
Also called 'bilinear' filtering
(if you set the magfilter to this too!)

More Minification filters

- By choosing the nearest mipmap (*_MIPMAP_NEAREST)



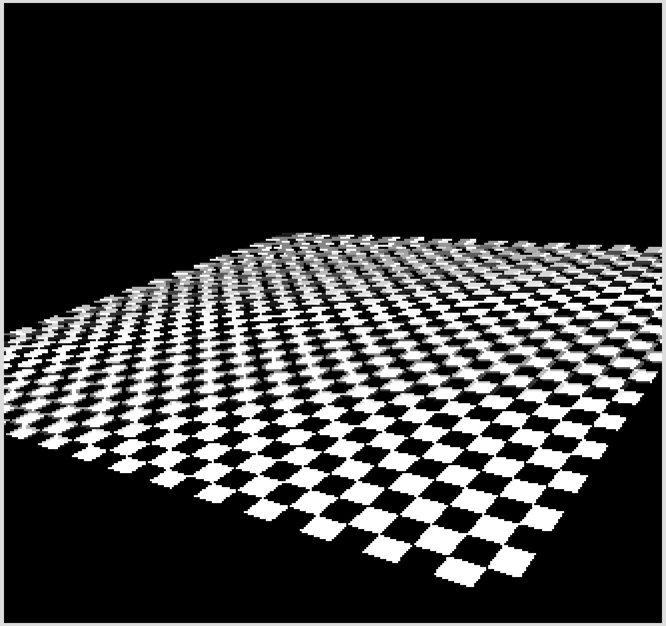
GL_NEAREST_MIPMAP_NEAREST
Fastest choice, not pretty, visible 'banding'



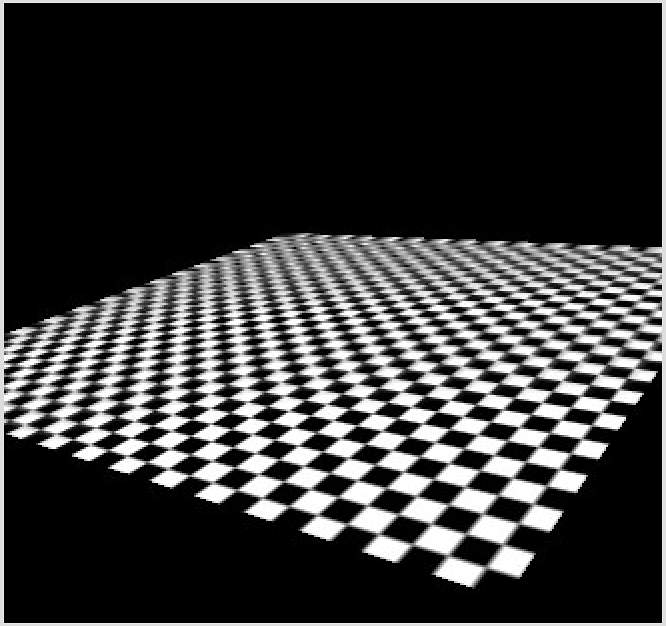
GL_LINEAR_MIPMAP_NEAREST
Very visible 'banding', quite fast

More Minification filters

- By interpolating the nearest mipmaps (*_MIPMAP_LINEAR)



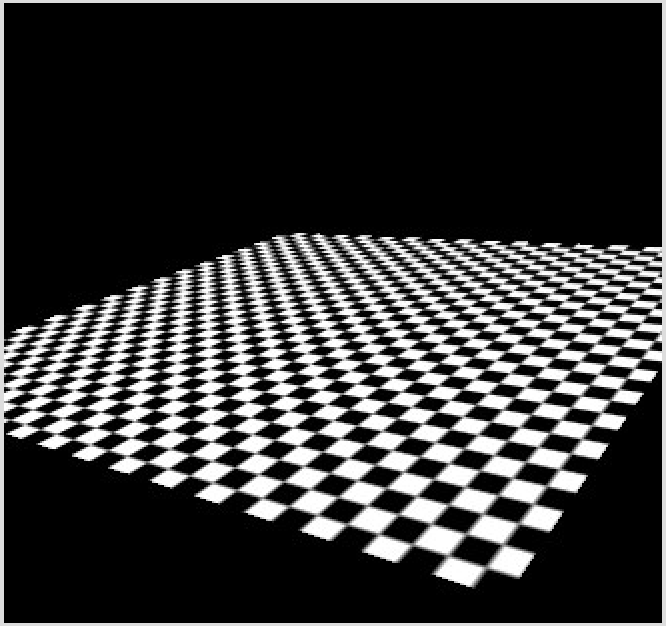
GL_NEAREST_MIPMAP_LINEAR
Default setting in OpenGL (!)
Not pretty for the chessboard
Far distance turns into 'grey goo'
Best speed vs quality



GL_LINEAR_MIPMAP_LINEAR
High quality, somewhat expensive
Far distance turns into 'grey goo'
Also called 'trilinear filtering'

Anisotropic Filtering

- Special filter available through extension



Anistropic filtering
Very nice adjustable quality
Relatively expensive

Texture Object Properties

(mipmaps included)

- Per-mipmap
  - Width and height -Power of two?
  - Data Format -RGB8, RGBA8, +++
  - Border -Usually 0
  - The texel data itself
- Per texture object
  - Dimensionality -1D, 2D, 3D or Cube
  - Wrapping rules -Clamp or Repeat
  - Minification and Magnification Filter

Setting per-mipmap properties ...

- glTexImage2D(

```
target            - GL_TEXTURE_2D
miplevel          - 0 through whatever
internalformat    - GL_RGB, GL_RGBA
width
height
border            - typically 0
format   \
datatype  >       - input parameters
pointer  /
);
```

... and filtermodes ...

- glTexParameteri(

```
target    - GL_TEXTURE_2D
pname     - GL_TEXTURE_MIN_FILTER
          - GL_TEXTURE_MAG_FILTER
value     - GL_LINEAR
          - GL_NEAREST
          - GL_*_MIPMAP_LINEAR
          - GL_*_MIPMAP_NEAREST
);
```

... and wrapmodes!

- glTexParameteri(

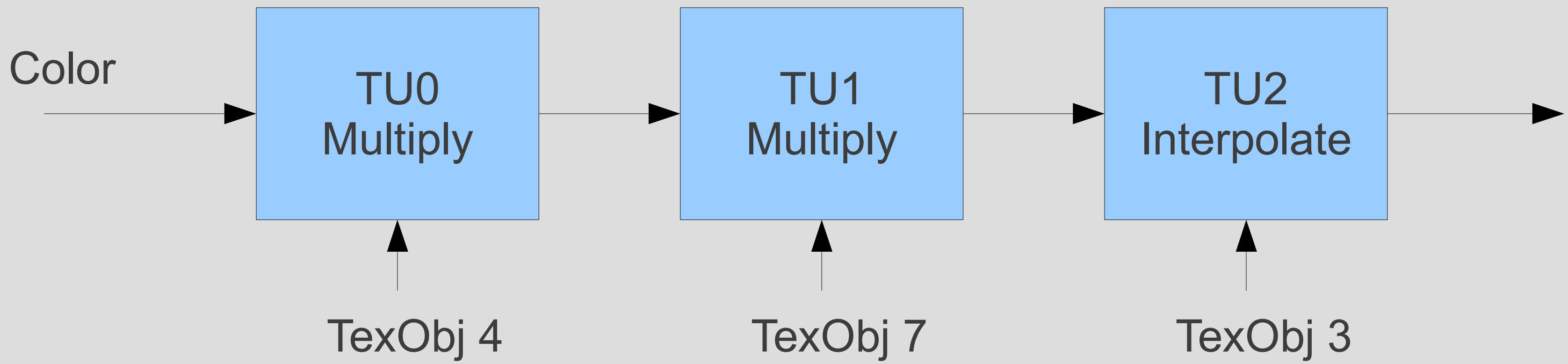
```
target    - GL_TEXTURE_2D
pname     - GL_TEXTURE_WRAP_S
          - GL_TEXTURE_WRAP_T
value     - GL_REPEAT
          - GL_CLAMP
);
```

Enough theory!

- Let's do some texture examples

Texture Units

- OpenGL supports multitexturing
  - Up to 8 texture units at the same time



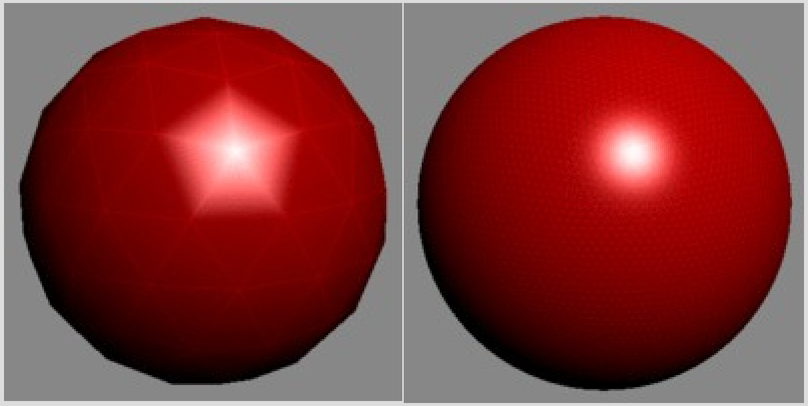
- glActiveTexture / glClientActiveTexture
- Very very annoying to use
  - Ignore these, use shaders ;)

Tips and Tricks on Texturing

- OpenGL will swap textures in and out of GPU mem on demand
  - This happens on glBindTexture(...)
- Envmapped textures are easy eyecandy
  - We'll do that later on
- Multitexturing – don't go there w/o shaders

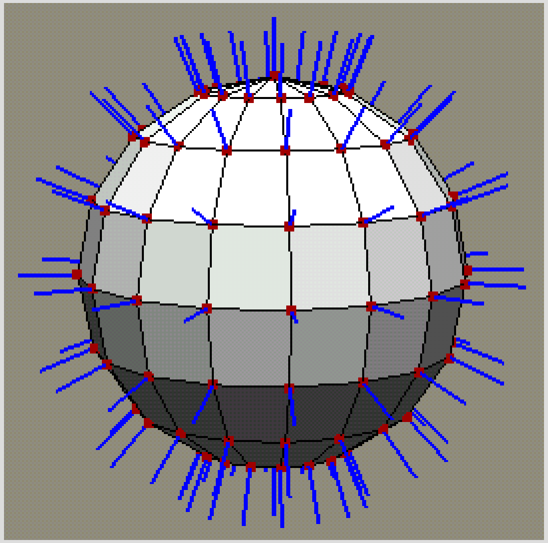
OpenGL Lighting

- 2 types of lighting
  - Per-vertex lighting
  - Per-pixel lighting (require shaders)
- Gouraud and Phong
  - Identical per-vertex and per-pixel light models
  - Alter the color of each vertex based on
    - Known Light sources
    - Ambient Light
    - Surface properties (Materials)



Normals

- Each polyon has two faces
  - Front side
  - Back side
- The normal decides which face is 'front'
  - One unit long
- More importantly:
  - Normal is useful in lighting calculations



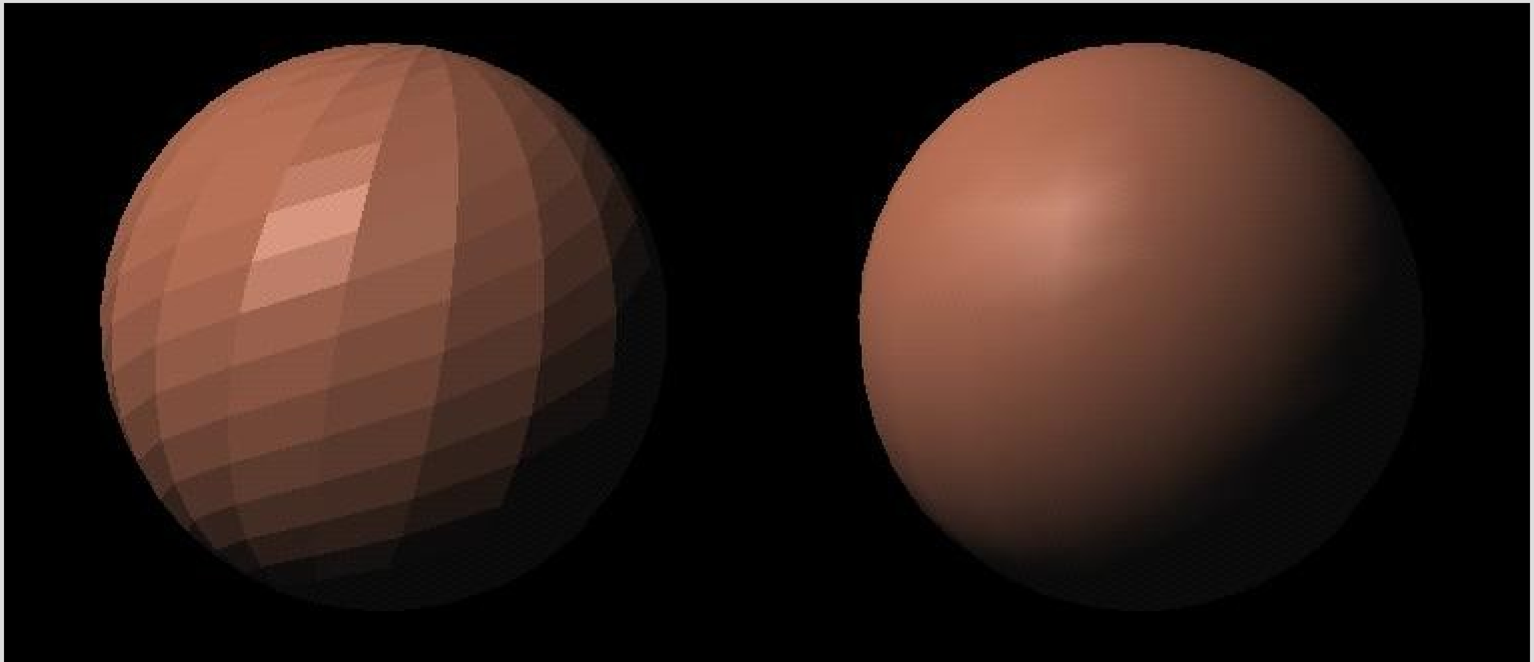
Specifying Normals

- The OpenGL lighting model require Normals
  - Can be calculated, but with some limitations...
  - Typically provided by 3Dstudio
- glNormalPointer(...)
  - Works like all the other pointer functions
  - Like color, a normal is a vertex attribute

Facenormals vs vertexnormals

- A normal is a face attribute
- OpenGL works with *vertex* attributes
  - This is actually better!

- Flat faces vs smooth faces



Specifying Light Sources

- OpenGL fixed function T&L supports 8 lights
  - If you need more, create a system which selects the 8 most significant ones
- Each light source has a
  - Position - 'world coordinates'
  - Diffuse/Ambient color - usually the same
  - Specular color
  - Direction/Cone-angle - if a spotlight
- Use glLightfv to specify all this

Phong/Gourard Light Model

$$I_p = k_a i_a + \sum_{\mathrm{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s) + k_e.$$

- Ambient Light - constant background lighting
- Diffuse Light - light reflected from surfaces
- Specular Light - light reflected from shiny surfaces
- Emissive Light - glowing light
  - Phong/Gourard does not permit surfaces to enlighten eachother

Ambient

Ambient+Diffuse

Ambient+Diffuse+Specular

Ambient+Diffuse+Specular+Emission

Phong/Gourard Light Model

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s) + k_e.$$

- Ambient Light is constant
- Diffuse Light is simply dot-multiplied with the normal
- Specular light is dot-multiplied with the view angle
  - And taken into a power of *alpha*

What determine materials?

$$I_p = k_a i_a + \sum_{\text{lights}} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s) + k_e.$$

- The alpha decides the 'shinyness' of the *material*
  - OpenGL: between 0 (hard) an 128 (virtually invisible)

- Ka and kd are usually identical
  - Typically the color of the object
  - Since everything is usually textured, normally white
- Ks is the shinyness color of the material
  - Usually white for metallic or plastic surfaces
- Ke is very rarely used, usually zero.

Lights

- Enough theory, let's do an example!
  - Per-vertex lighting
  - Per-pixel lighting

Vertex Buffer Objects

- Sending pointers per drawcall is not optimal
  - Buses not suited for bursts of small data packets
- Better solution:
  - pre-upload vertex data to GPU
- Vertex Buffer Objects (VBO's)

Types of VBOs

- STATIC - Non-skinned objects
- DYNAMIC - Skinned objects
- STREAM - To be used once

- DRAW - data only used for drawing
- READ - data only used for reading
- COPY - both draw and read

- Turns into these enums:
  - GL_STATIC_READ
  - etc

VBOs are very easy to use

- glGenBuffers
- glBindBuffer
- glBufferData(

```
target      - GL_ARRAY_BUFFER or
              GL_ELEMENT_ARRAY_BUFFER
size        - bytesize of this buffer
ptr         - data to put in buffer. Or NULL
type        - enum from last slide
);
```

- Can be mapped
  - glMapBuffer / glUnmapBuffer

VBO Example?

- Well, okay...

OpenGL: The Bigger Picture

- Models come from 3D studio or Blender
  - Rarely from hand-programmed arrays
- Each model have N drawcalls
- Each drawcall have one material
  - Diffuse Color, Texture
  - Specular Color, hardness
  - And often more – check 3Dstudio
- Ultimately, you want a model.draw()
  - Sets up materials
  - Calls the proper draw
- API does not really matter!

And finally... shaders?

- GLSL
  - C-like vectorbased shading language
- Programs
  - Vertex shader + Fragment Shader
  - Replace the fixed-function pipeline
    - Do everything yourself... ouch?
    - Great possibilities
- Maybe a later course ;)

Questions and stuff

- Fire away!