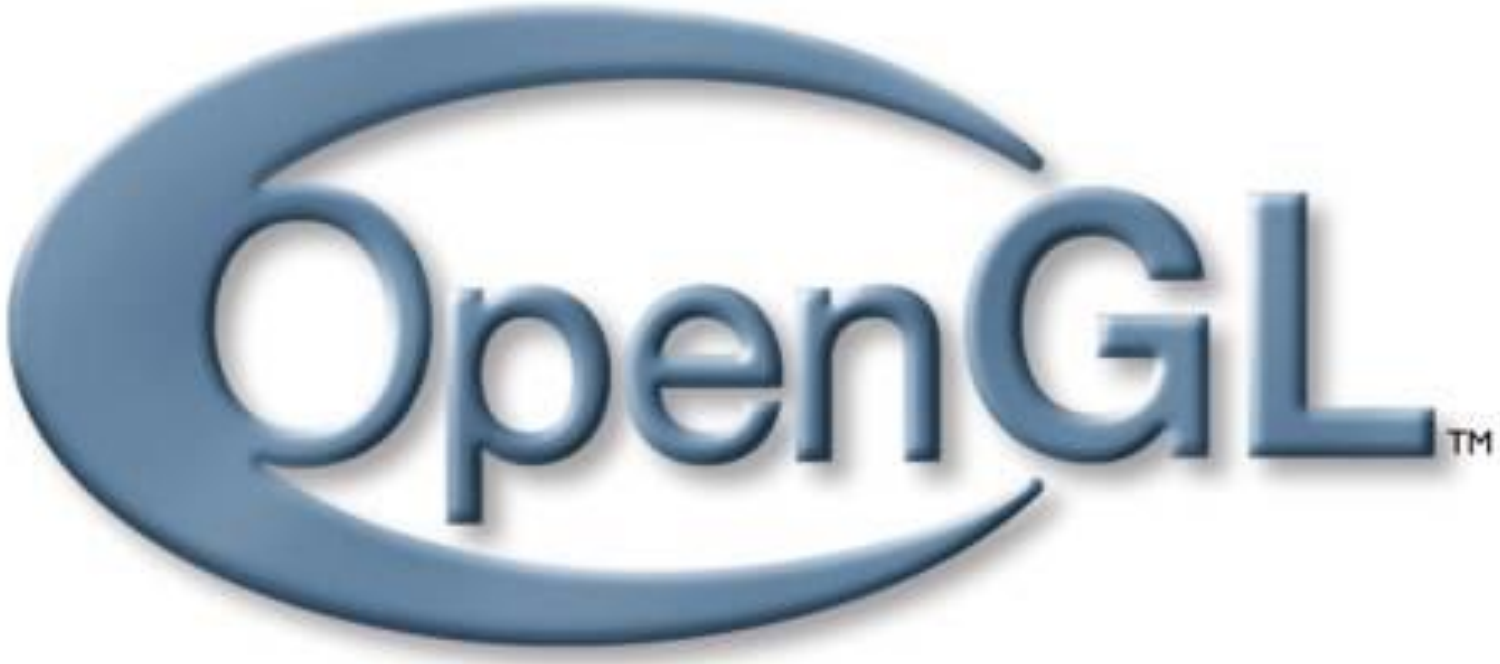


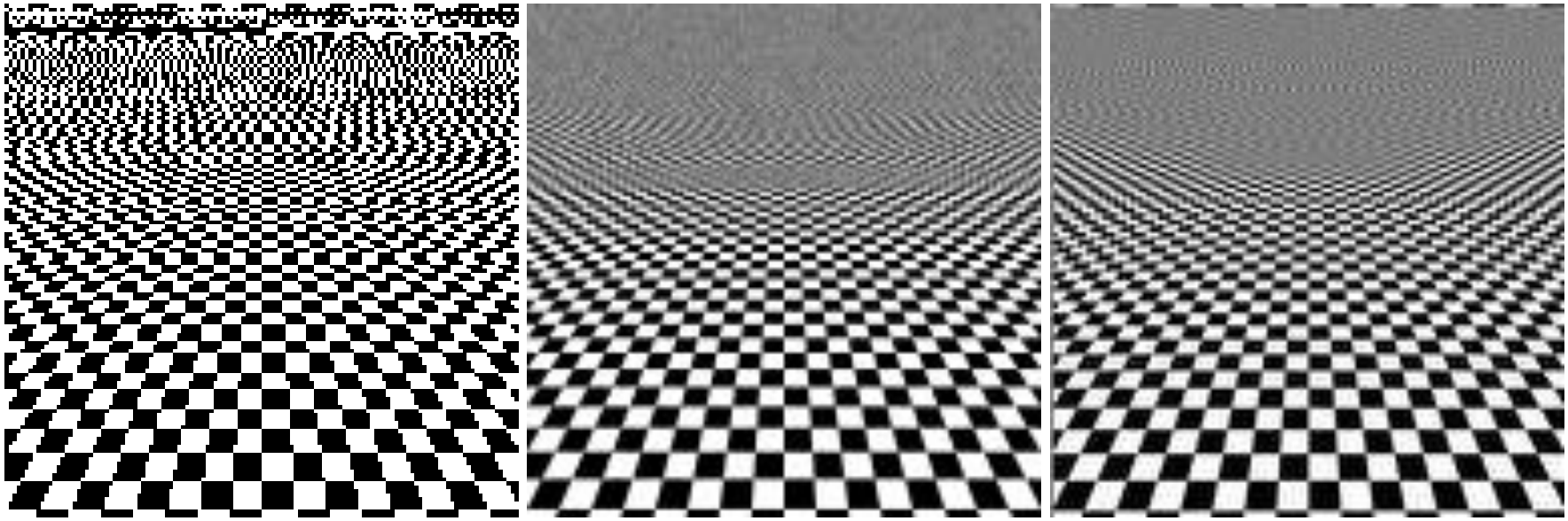
part 10

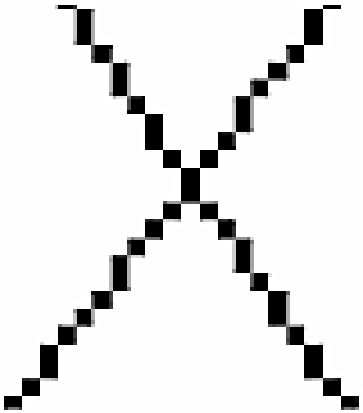
Martin Samuelčík
http://www.sccg.sk/~samuelcik
Room I4

Anti-aliasing

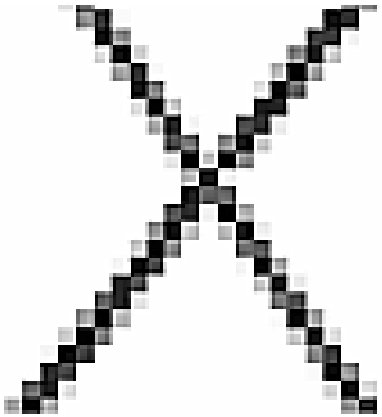
- Alias
  - Problem of insufficient sampling of intensity function
  - Jagged lines and polygons
  - Artifact in textures
- Anti-aliasing
  - Removing alias artifacts
  - Denser sampling
  - Lowering frequencies in functions

Anti-aliasing





Aliased



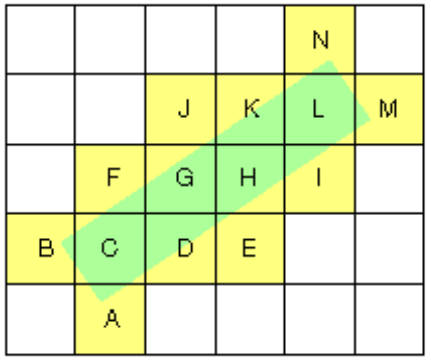
Antialiased

Anti-aliasing

- Mip-mapping
  - Lowering frequencies in textures
  - Using smaller versions of given texture
- Primitives anti-aliasing
  - Lowering frequencies on borders of primitives
  - Blurring edges of primitives during rasterization
- FSAA (Full-screen anti-aliasing)
  - Applying anti-aliasing on whole image
  - Denser sampling
  - Super sampling

Primitives anti-aliasing

- Computing pixel coverage values of primitives, combination with alpha values
- Exact computation and result are based on implementation
- **void glHint(GLenum target, GLenum hint)**
  - target = GL_POINT_SMOOTH_ GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT
  - hint = GL_FASTEST, GL_NICEST, GL_DONT_CARE
  - Setting desired quality of anti-aliasing



A .040510
B .040510
C .878469
D .434259
E .007639
F .141435
G .759952
H .759952
I .141435
J .007639
K .434259
L .878469
M .040510
N .040510

Primitives anti-aliasing

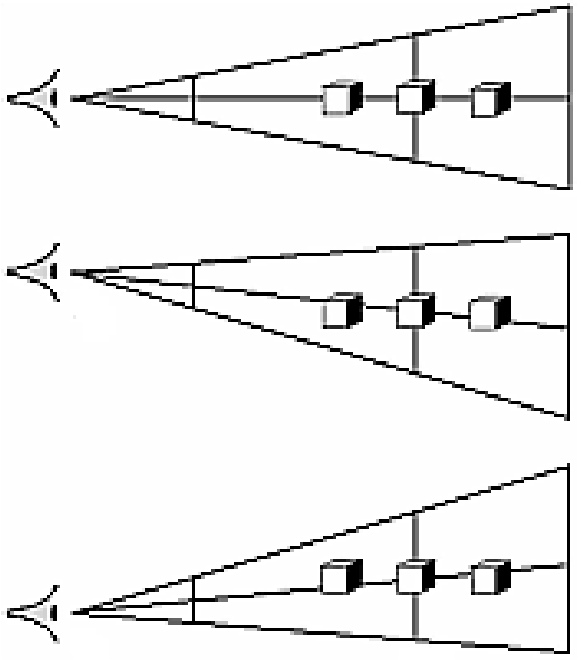
- Enabling/disabling for each type of primitive
- **glEnable(GL_POINT_SMOOTH);**
- **glEnable(GL_LINE_SMOOTH);**
- **glEnable(GL_POLYGON_SMOOTH);**
- Must use blending
- Coverage values used as blending alphas
- Back-to-front drawing or additive blending
- Not preferred technique

FSAA

- Anti-aliasing technique for whole screen
- Several algorithms working on whole screen independently on primitives
- Super sampling – rendering bigger image then screen, than downsampling
- Accumulation buffer – rendering scene for slightly different camera projections, than averaging renderings into final image
- Multi sampling – generating multiple fragments (samples) for each pixel in rasterizer

Accumulation buffer

- Used for accumulating several renderings - samples
- Each rendering has slightly different projection with same camera position



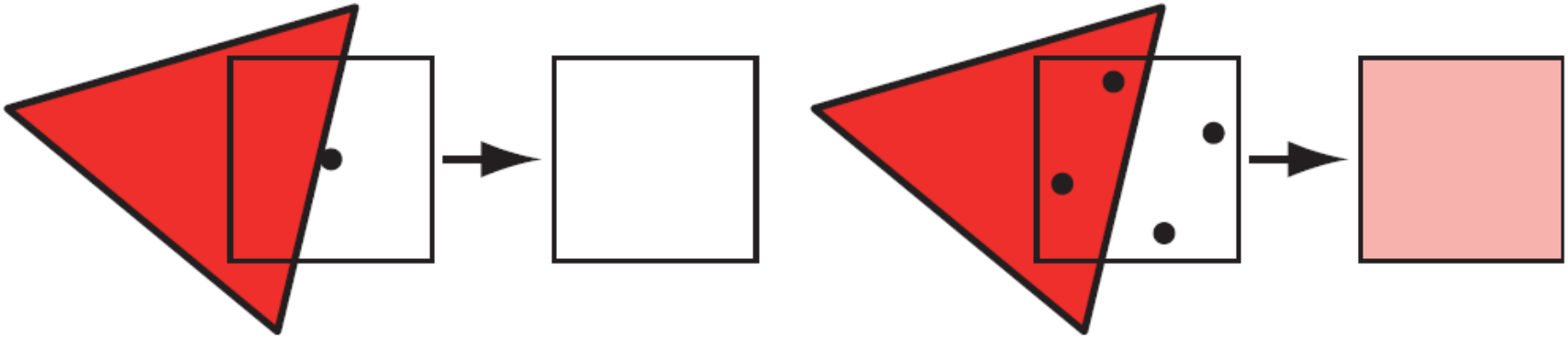
Accumulation AA

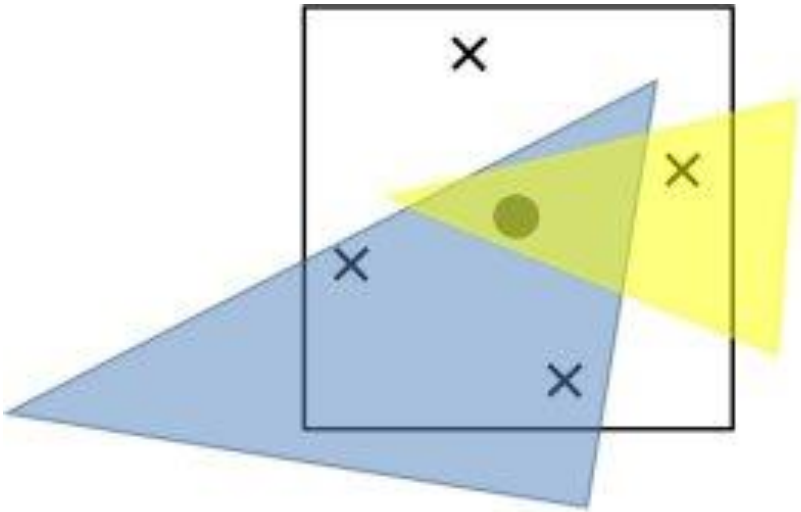
```
typedef struct {
   GLfloat x, y;
} jitter_point;
jitter_point j8[] ={ {-0.334818,  0.435331}, {0.286438, -0.393495}, {0.459462,  0.141540}, {-
      0.414498, -0.192829}, {-0.183790,  0.082102}, {-0.079263, -0.317383}, {0.102254,  0.299133},
      {0.164216, -0.054399}};

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_ACCUM_BUFFER_BIT);
GLint viewport[4];
glGetIntegerv(GL_VIEWPORT, viewport);
if (giNumSamples > 8) giNumSamples = 8;

for (jitter = 0; jitter < giNumSamples; jitter++)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   GLdouble dx = -j8[jitter].x * (cam_right - cam_left) / (GLdouble) viewport[2];
   GLdouble dy = -j8[jitter].y * (cam_top - cam_bottom) / (GLdouble) viewport[3];
   glFrustum(cam_left + dx, cam_right + dx, cam_bottom + dy, cam_top + dy, cam_near, cam_far);
   glMatrixMode(GL_MODELVIEW);
   displayScene();
   glAccum(GL_ACCUM, 1.0/giNumSamples);
}
glAccum(GL_RETURN, 1.0);
```

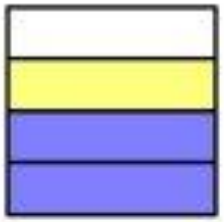
Multisample AA

- Multiple samples generated in rasterizer, used for visibility tests (depth/stencil), fragment shader is executed still once
- Final color is computed from fragment color multiplied by coverage (percentage of samples that passed depth/stencil tests)
- Depth and stencil buffers for each sample
- Pixel format with given number of samples for multisampling must be presented, chosen before OpenGL context is created, usually at the beginning of application

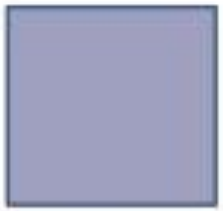
MSAA







Resulting samples



Pixel color after resolve

Color sample

Depth Sample

MSAA

MSAA example

- Opening OpenGL window with pixel format that support MSAA
- Framebuffer must be initialized with multiple depth/stencil buffers
- Heavy system dependent initialization – using support libraries – WGL for Win32 platform
- GLUT have MSAA initialization, no definition of number of samples
- OpenGL extension for multisample, in core from OpenGL 1.3, GL_ARB_multisample

Win32 OpenGL window

- Create Win32 window
  - HWND – identifier of window
  - HDC – identifier of device context
- Choose and set new pixel format for window
  - Pixel format exposes capabilities of system (operating system, driver, hardware)
  - Each pixel format has properties like, number of buffers, buffers precision, rendering acceleration, multisampling, …
- Create OpenGL context
  - OpenGL context is state machine that holds state variables and manages OpenGL functions
  - Only after context is created, application can call OpenGL functions

No MSAA – Win32 example

```
// Description of pixel format
PIXELFORMATDESCRIPTOR pfd =
   {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER,   //Flags
      PFD_TYPE_RGBA,            //The kind of framebuffer. RGBA or palette.
      32,                       //Colordepth of the framebuffer.
      0, 0, 0, 0, 0, 0,
      0,
      0,
      0,
      0, 0, 0, 0,
      24,                       //Number of bits for the depthbuffer
      8,                        //Number of bits for the stencilbuffer
      0,                        //Number of Aux buffers in the framebuffer.
      PFD_MAIN_PLANE,
      0,
      0, 0, 0
   };
```

No MSAA – Win32 example

```
// define and register window class for our OpenGL window
WNDCLASS wc = {0};
wc.lpfnWndProc = WndProc;
wc.hInstance     = hInstance;
wc.hbrBackground = (HBRUSH)(COLOR_BACKGROUND);
wc.lpszClassName = L"oglversionchecksample";
wc.style = CS_OWNDC;
RegisterClass (&wc);

// create window based on defined class
HWND hWnd = CreateWindow (wc.lpszClassName, L"openg lwindow", WS_OVERLAPPEDWINDOW | WS_VISIBLE, 0, 0,
      640, 480, 0, 0, hInstance, 0);

// getting handle to device context from identifier of window
HDC ourWindowHandleToDeviceContext = GetDC (hWnd);

// get id of existing pixel format that is close match to supplied pfd
int letWindowsChooseThisPixelFormat;
letWindowsChooseThisPixelFormat = ChoosePixelFormat (ourWindowHandleToDeviceContext, &pfd);

// set chosen pixel format for our window and for OpenGL context creation
SetPixelFormat (ourWindowHandleToDeviceContext,letWindowsChooseThisPixelFormat, &pfd);

// create OpenGL context for our window
HGLRC ourOpenGLRenderingContext = wglCreateContext (ourWindowHandleToDeviceContext);

// set created context as current
wglMakeCurrent (ourWindowHandleToDeviceContext, ourOpenGLRenderingContext);
// from now on, context is created and activated and we can call OpenGL functions
```

MSAA – Win32 example

```
// create temporary context so we can call functions from WGL extensions
int nPixelFormat = ChoosePixelFormat (ourWindowHandleToDeviceContext, &pfd);
SetPixelFormat (ourWindowHandleToDeviceContext, nPixelFormat, &pfd);
HGLRC tempContext = wglCreateContext (ourWindowHandleToDeviceContext);
wglMakeCurrent (ourWindowHandleToDeviceContext, tempContext);

// fill attributes of pixel format that we want to use within our window
int attributes[] = {
   WGL_DRAW_TO_WINDOW_ARB, GL_TRUE,
   WGL_SUPPORT_OPENGL_ARB, GL_TRUE,
   WGL_DOUBLE_BUFFER_ARB, GL_TRUE,
   WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB,
   WGL_COLOR_BITS_ARB, 32,
   WGL_DEPTH_BITS_ARB, 24,
   WGL_STENCIL_BITS_ARB, 8,
   WGL_SAMPLE_BUFFERS_ARB, 1, // Enable multisampling
   WGL_SAMPLES_ARB, 8,        // Number of samples
   0};

// check if such pixel format is present in our system
int pixelFormatMultisample;
UINT numFormats;
float fAttributes[] = {0,0};
bool valid = wglChoosePixelFormatARB (hDC, attributes, fAttributes, 1, &pixelFormatMultisample, &numFormats);
if (valid && numFormats >= 1)
{
   // pixel format with multisampling have been found, create new OpenGL context for it
   wglMakeCurrent (NULL, NULL);
   wglDeleteContext (tempContext);
   SetPixelFormat (ourWindowHandleToDeviceContext, pixelFormatMultisample, &pfd);
   HGLRC multisampleContext = wglCreateContext (ourWindowHandleToDeviceContext);
   wglMakeCurrent (ourWindowHandleToDeviceContext, multisampleContext);
   glEnable (GL_MULTISAMPLE);
}
```

Fog

- Mixing of fragment color and fog color
- Mixing factor is based on depth of fragment, height of fragment in world, …
- Objects fully in fog not rendered at all

OpenGL fog

- Setting parameters of fog
  - **void glFog{if}(GLenum pname, TYPE param)**
  - **void glFog{if}v(GLenum pname, TYPE *params)**
- Turing fog on/off
  - **glEnable(GL_FOG)**
  - **glDisable(GL_FOG)**
- When using shaders, this type of fog is disabled and replaced

OpenGL fog parameters

pname	param(s)
GL_FOG_MODE	GL_EXP (the default), GL_EXP2, or GL_LINEAR
GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END	*density*, *start*, or *end* in the linear and exp equations (start and end in world distance)
GL_FOG_COLOR	fog's RGBA color values
GL_FOG_INDEX	fog's index color values

OpenGL fog

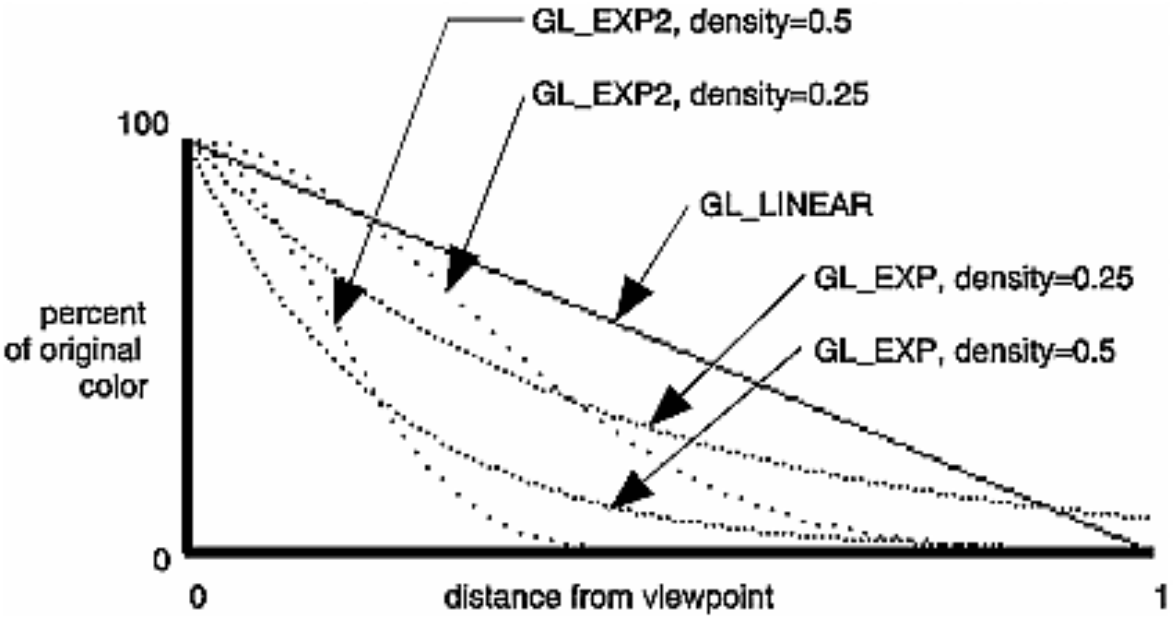
- z – depth of fragment, from <0,1>

$$f = e^{-(density \cdot z)} \text{ (GL\_EXP)}$$

$$f = e^{-(density \cdot z)^2} \text{ (GL\_EXP2)}$$

$$f = \frac{end - z}{end - start} \text{ (GL\_LINEAR)}$$

$$C_{final} = f\, C_{fragment} + (1 - f)\, C_{fog}$$



OpenGL fog example

```
GLfloat fogColor[4]= {0.5f, 0.5f, 0.5f, 1.0f};
glFogi(GL_FOG_MODE, GL_EXP2);
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_DENSITY, 0.35f);
glHint(GL_FOG_HINT, GL_DONT_CARE);
glFogf(GL_FOG_START, 1.0f);
glFogf(GL_FOG_END, 5.0f);
glEnable(GL_FOG);
```

GLSL fog

- Simply in fragment shader, combine fog color and color from illumination
- For mixing factor, use depth of fragment stored in gl_FragCoord.z
- gl_FragCoord.z is from normalized space, use gl_FragCoord.z / gl_FragCoord.w to get measurement in world space
- Additional parameters can be used for mixing, like height of fragment in

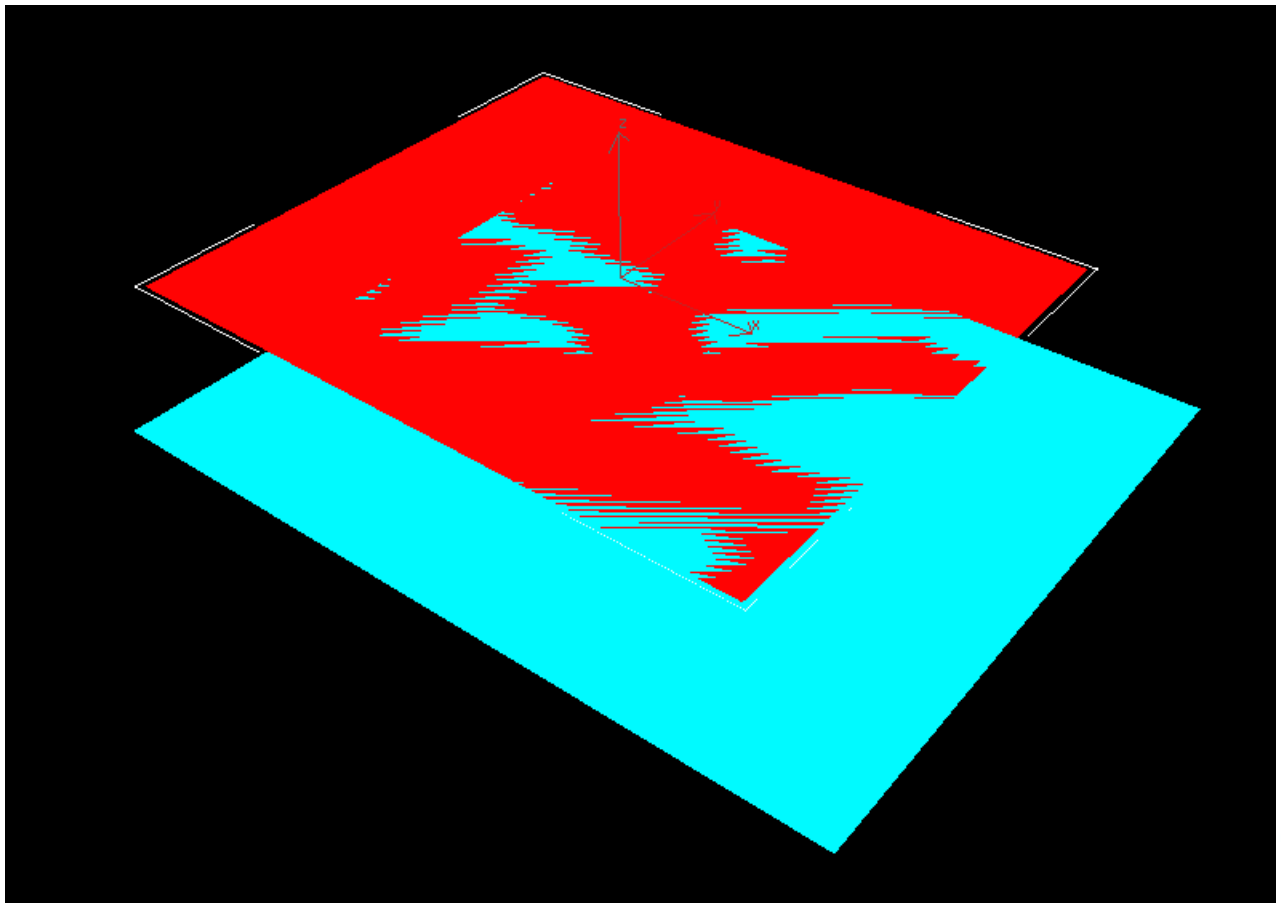
GLSL fog example

```
// FRAGMENT SHADER FOR FOG
uniform sampler2D color_texture;
uniform vec4 fog_color;
uniform float max_fog_height;
uniform float fog_start;
uniform float fog_end;

varying float height;

void main()
{
   // compute factor based on height of fragment in world
   float ratio_height = height / max_fog_height;
   ratio_height = clamp(ratio_height, 0, 1);

   // compute factor based on depth of fragment in camera
   float depth_world = gl_FragCoord.z / gl_FragCoord.w;
   float ratio_depth = (fog_end - depth_world) / (fog_end – fog_start);
   ratio_depth = clamp(ratio_depth, 0, 1);

   // combine color of fog and color from texture
   float fog_factor = ratio_height * ratio_depth;
   mix(texture2D(color_texture, gl_TexCoord[0].st), fog_color, fogFactor);
}
```

```
// VERTEX SHADER FOR FOG
uniform mat4 view_matrix;

varying float height;

void main()
{
   // get height of vertex in world space
   vec4 V_world = gl_ModelViewMatrix * gl_Vertex;
   height = v_world.z;

   gl_TexCoord[0] = gl_MultiTexCoord0;
   gl_Position = gl_ProjectionMatrix * view_matrix * V_world;
}
```

Z-fighting

- Artifacts when rendering coplanar conjunctive polygons



Polygon offset

- Removing z-fighting artifacts
- Adding some small amount to z-coordinate in screen space – similar to slightly moving polygon away from or towards to camera
- Polygon have same position, depth of generated fragments is altered
- Added offset is computed as linear combination of polygon slope and precision in depth values
- Or use shaders ☺

Polygon offset

- Enable, disable for each type of primitives
  - **glEnable(GL_POLYGON_OFFSET_FILL)**
  - **glEnable(GL_POLYGON_OFFSET_LINE)**
  - **glEnable(GL_POLYGON_OFFSET_POINT)**
- **void glPolygonOffset(GLfloat factor, GLfloat units)**
  - Set how the offset is calculated
  - Offset calculation: o = m*factor + r*units
    - m - maximum depth slope of polygon
    - r - smallest value to produce a resolvable difference in window coordinate depth values

Polygon offset example

```
// RENDER OBJECT TOGETHER WITH ITS WIREFRAME

// RENDER OBJECT WITH POLYGON OFFSET, FILLED OBJECT HAS DEPTH
    VALUES SLIGHTLY MOVED AWAY FROM CAMERA
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 1.0);
renderObject();
glDisable(GL_POLYGON_OFFSET_FILL);

// RENDER WIREFRAME OBJECT WITHOUT POLYGON OFFSET
glDisable(GL_LIGHTING);
glDisable(GL_LIGHT0);
glColor3f(1.0, 1.0, 1.0);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
renderObject();
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

The End!

Questions?