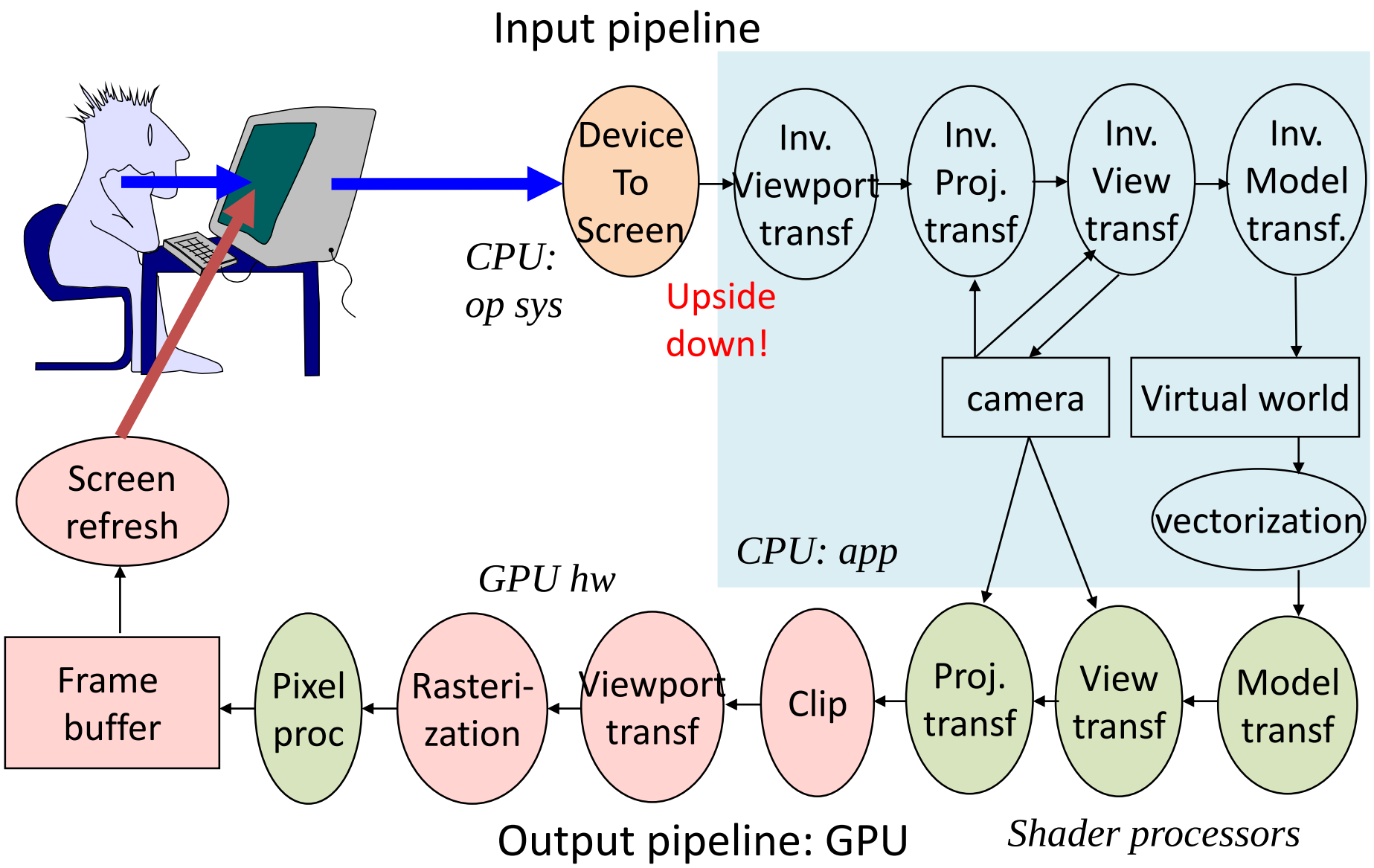
”In theory, there is no difference between theory and practice. In practice, there is.”

Benjamin Brewster

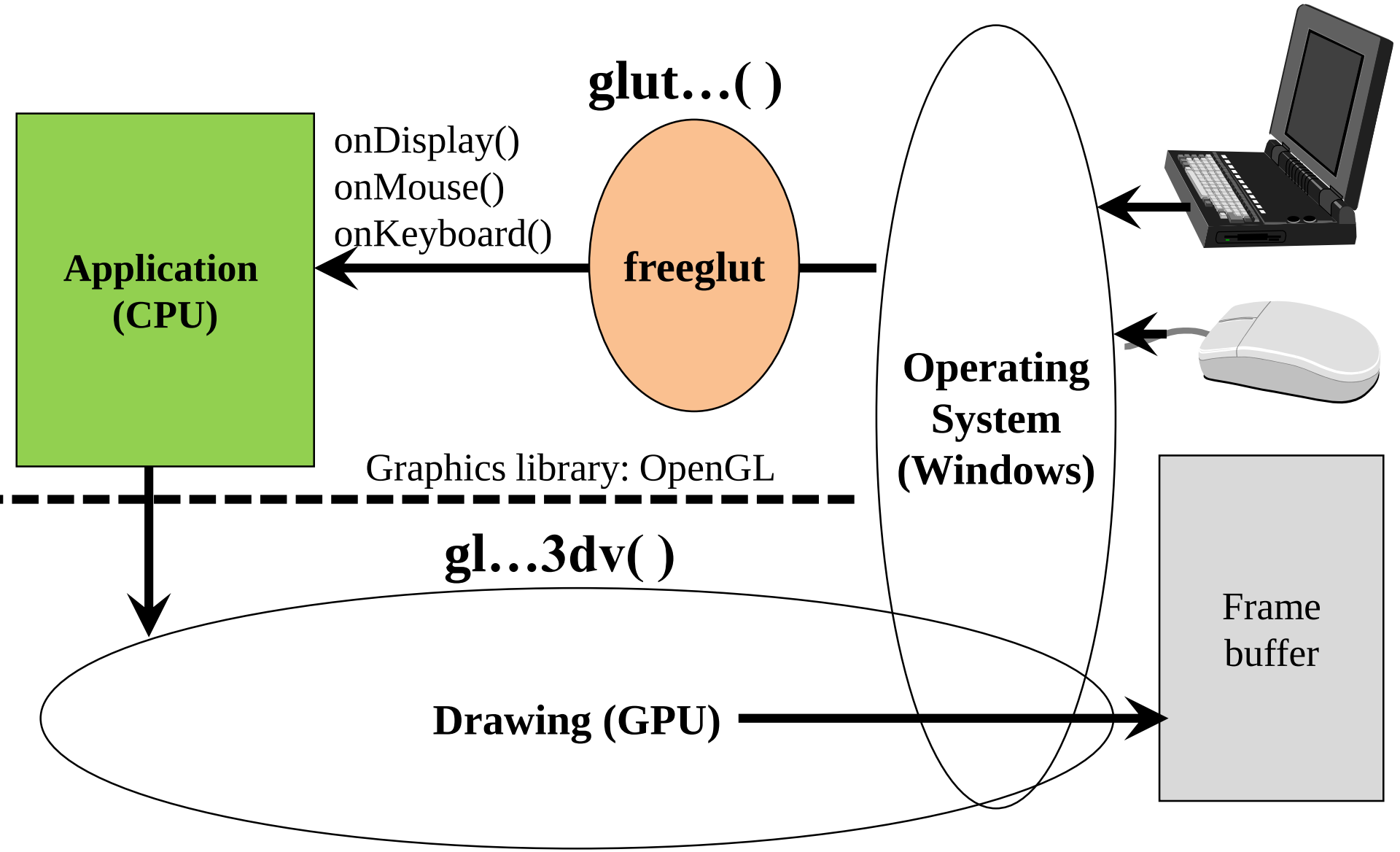
Graphics hardware and software

Szirmay-Kalos László

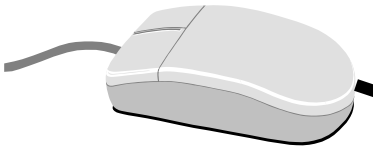
Interactive graphics systems

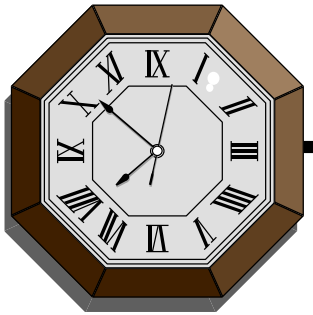


Software architecture



Event handling (GLUT)





Operating system

AppWindow setup
Callback registration

freeglut

callbacks

main

DisplayFunc

KeyboadFunc

KeyboadUpFunc

SpecialFunc

Reshape

MouseFunc

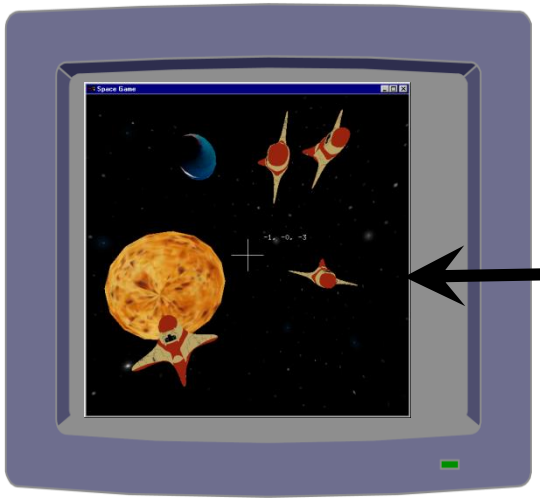
MotionFunc

IdleFunc

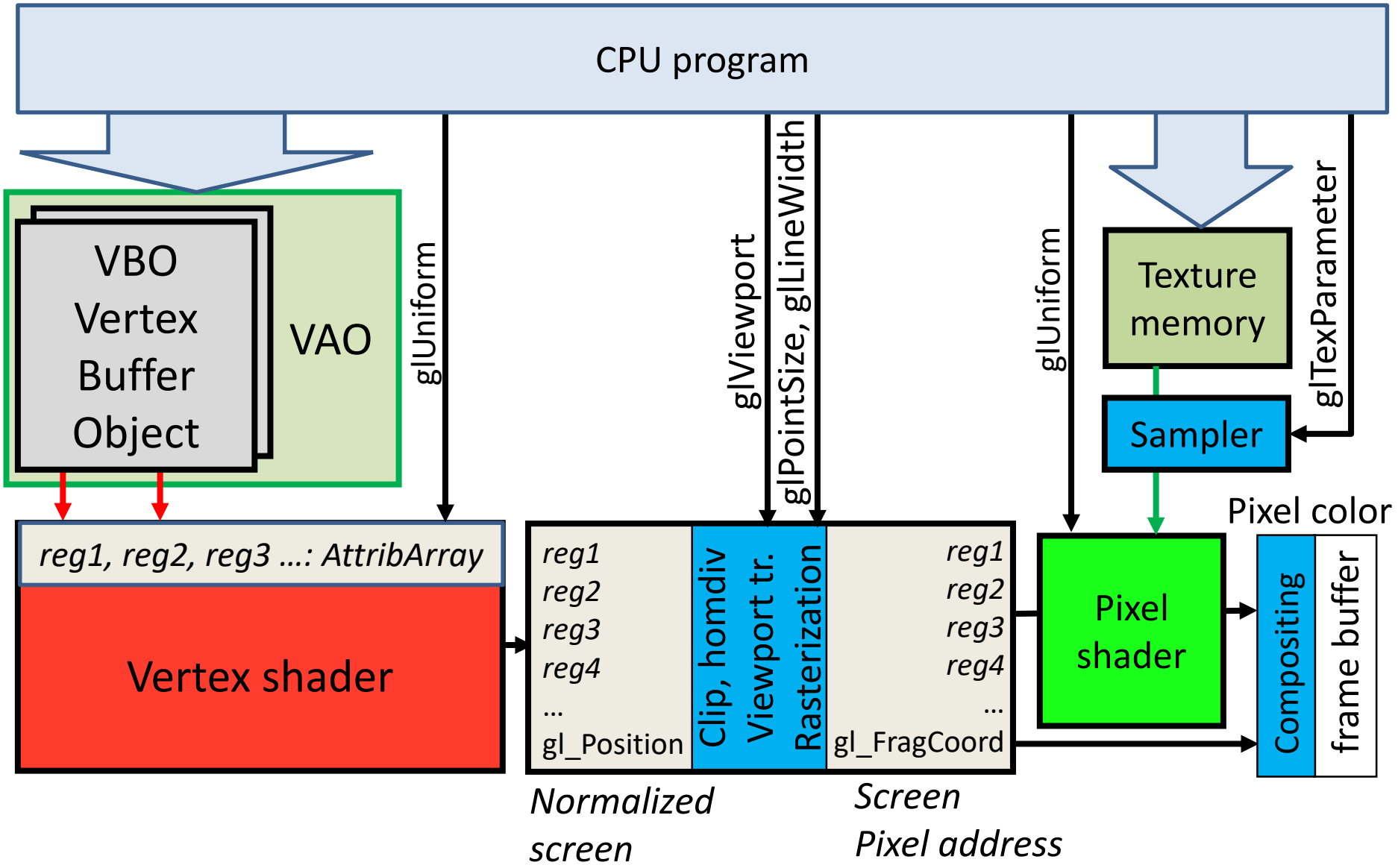
application

OpenGL

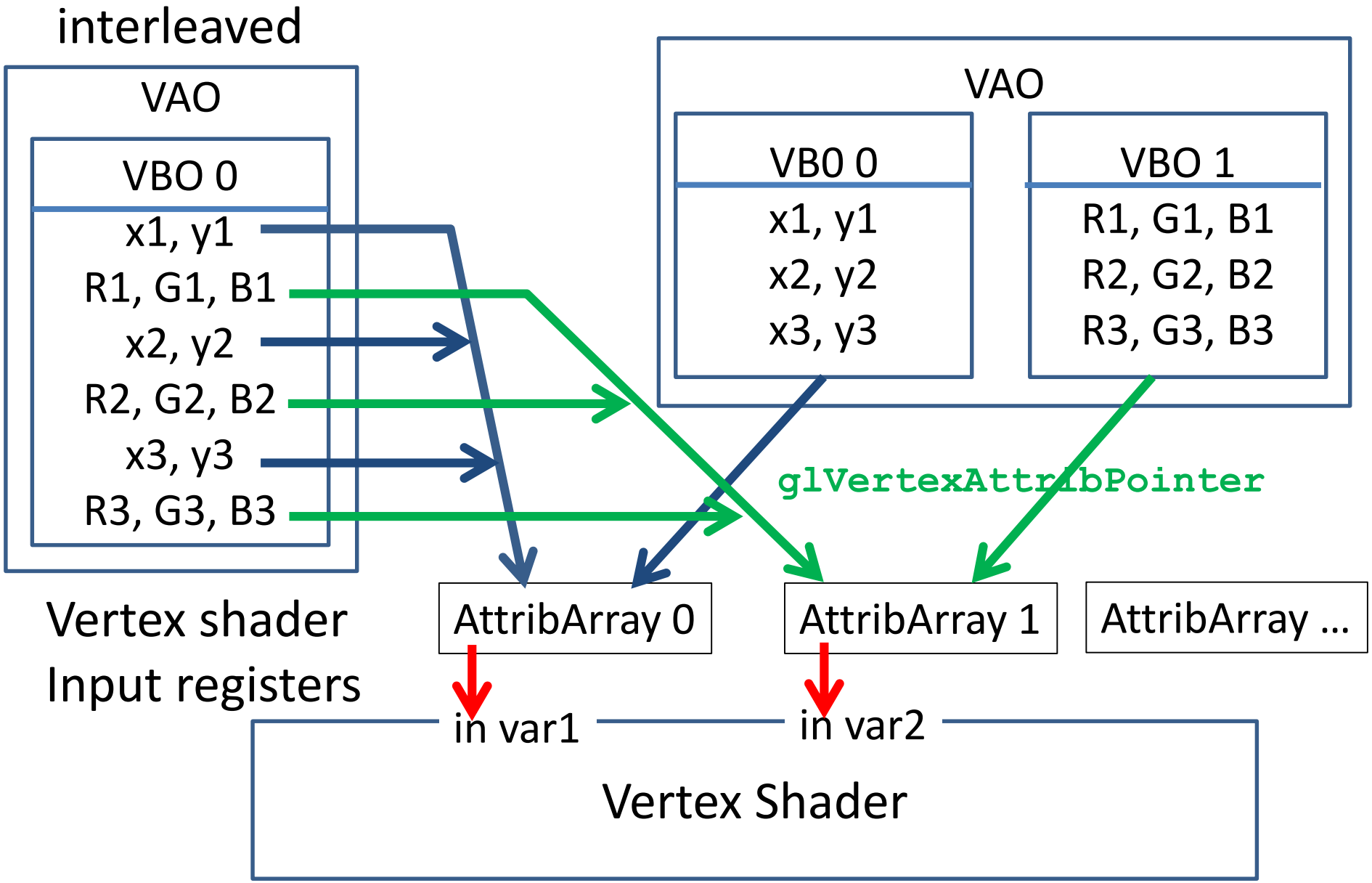
Graphics hardware



OpenGL 3.3 … 4.6

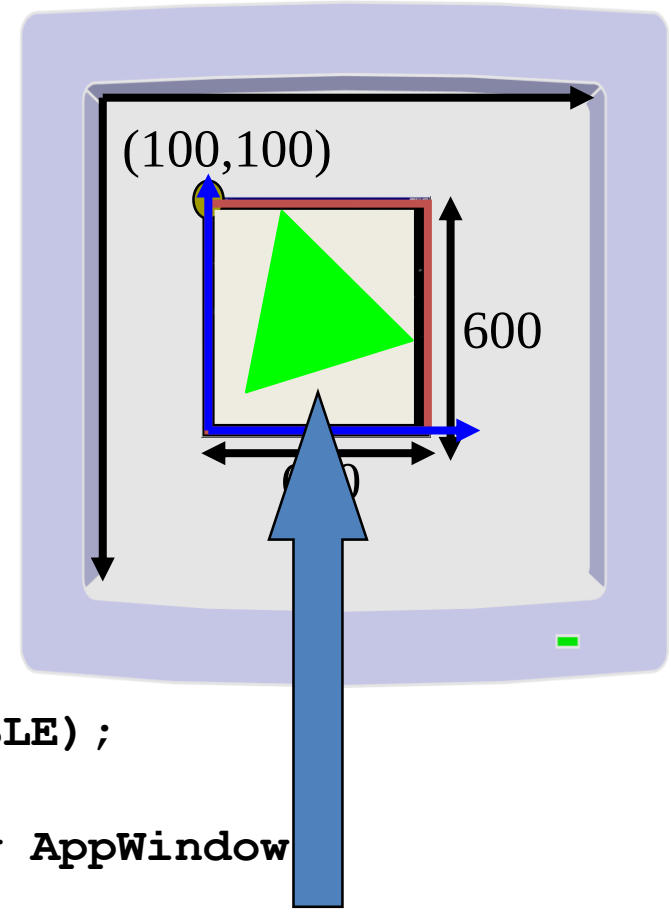

CPU program
VBO
Vertex
Buffer
Object
VAO
glUniform
glViewport
glPointSize, glLineWidth
glUniform
Texture
memory
glTexParameter
Sampler
Pixel color
reg1, reg2, reg3 …: AttribArray
Vertex shader
reg1
reg2
reg3
reg4
…
gl_Position
Clip, homdiv
Viewport tr.
Rasterization
reg1
reg2
reg3
reg4
…
gl_FragCoord
Pixel
shader
Compositing
frame buffer
Normalized
screen
Screen
Pixel address

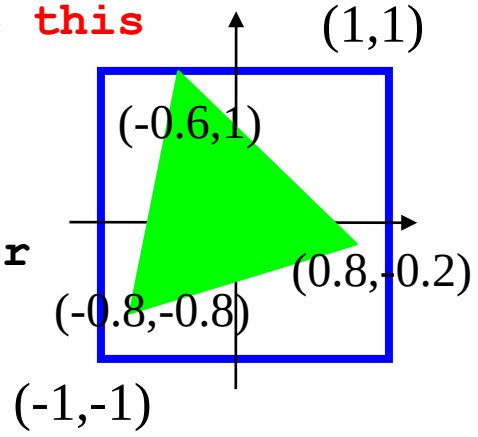
Vertex data streaming



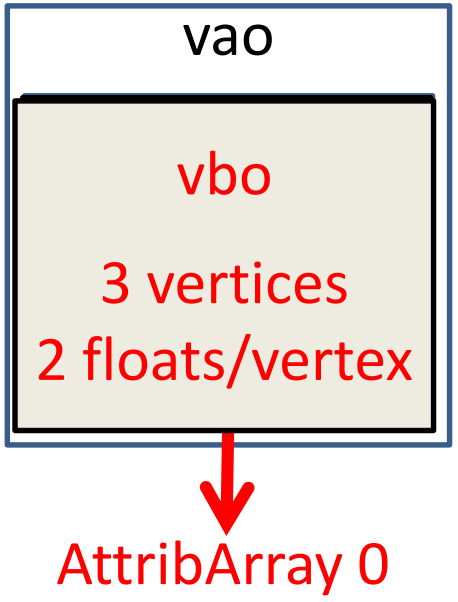
My first OpenGL program

```
#include <windows.h>    // Only in MsWin
#include <GL/glew.h>    // download
#include <GL/freeglut.h> // download

int main(int argc, char * argv[]) {
   glutInit(&argc, argv); // init glut
   glutInitContextVersion(3, 3);
   glutInitWindowSize(600, 600);
   glutInitWindowPosition(100, 100);
   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE);

   glutCreateWindow("Hi Graphics"); //show AppWindow
   glewExperimental = true; // magic
   glewInit(); // init glew, no opengl before this

   glViewport(0, 0, 600, 600); //photo
   onInitialization(); // next slide

   glutDisplayFunc(onDisplay); //event handler

   glutMainLoop();
   return 1;
}
```





onInitialization()




```
unsigned int shaderProgram;
unsigned int vao; // virtual world on the GPU

void onInitialization() {
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao); // make it active

    unsigned int vbo;// vertex buffer object
    glGenBuffers(1, &vbo); // Generate 1 buffer
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    // Geometry with 24 bytes (6 floats or 3 x 2 coordinates)
    float vertices[] = {-0.8,-0.8,  -0.6,1.0,  0.8,-0.2};
    glBufferData(GL_ARRAY_BUFFER,  // Copy to GPU target
                 sizeof(vertices), // # bytes
                 vertices,         // address
                 GL_STATIC_DRAW);  // we do not change later
    glEnableVertexAttribArray(0);  // AttribArray 0
    glVertexAttribPointer(0, // vbo -> AttribArray 0
      2, GL_FLOAT, GL_FALSE, // two floats/attrib, not fixed-point
      0, NULL);              // stride, offset: tightly packed
```

```
#version 330
precision highp float;
uniform mat4 MVP;
layout(location = 0) in vec2 vp; //AttribArray 0

void main() {
  gl_Position = vec4(vp.x,vp.y,0,1) * MVP;
}
```

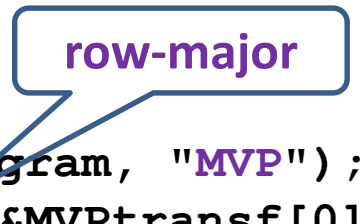
```
#version 330
precision highp float;
uniform vec3 color;
out vec4 outColor;

void main() {
   outColor = vec4(color,1);
}
```

```
    static const char * vertexSource = R"( … )";
    static const char * fragmentSource = R"( … )";
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexSource, NULL);
    glCompileShader(vertexShader);

    unsigned int fragmentShader=glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
    glCompileShader(fragmentShader);

    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);

    glBindFragDataLocation(shaderProgram, 0, "outColor");

    glLinkProgram(shaderProgram);
    glUseProgram(shaderProgram);
}
```

```
uniform mat4 MVP;
layout(location = 0) in vec2 vp;

void main() {
  gl_Position = vec4(vp.x,vp.y,0,1) * MVP;
}
```

```
uniform vec3 color;
out vec4 outColor;

void main() {
   outColor = vec4(color,1);
}
```

```
void onDisplay( ) {
    glClearColor(0, 0, 0, 0);      // background color
    glClear(GL_COLOR_BUFFER_BIT); // clear frame buffer

    // Set vertexColor to (0, 1, 0) = green
    int location = glGetUniformLocation(shaderProgram, "color");
    glUniform3f(location, 0.0f, 1.0f, 0.0f); // 3 floats

    float MVPtransf[4][4] = { 1, 0, 0, 0,     // MVP matrix,
                              0, 1, 0, 0,     // row-major!
                              0, 0, 1, 0,
                              0, 0, 0, 1 };

    location = glGetUniformLocation(shaderProgram, "MVP");
    glUniformMatrix4fv(location, 1, GL_TRUE, &MVPtransf[0][0]);

    glBindVertexArray(vao);  // Draw call
    glDrawArrays(GL_TRIANGLES, 0 /*startIdx*/, 3 /*# Elements*/);

    glutSwapBuffers( ); // exchange buffers for double buffering
}
```



OpenGL primitives

```
glDrawArrays( primitiveType,
              startIdx,
              numOfElements);
```

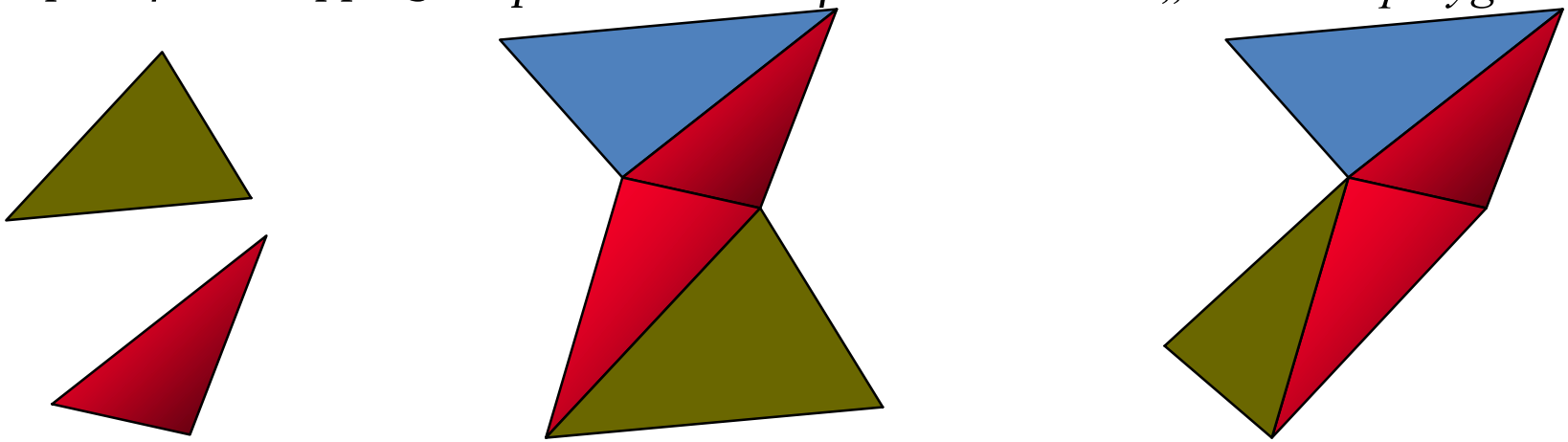
Vectorized parametric curve

GL_POINTS GL_LINES GL_LINE_STRIP GL_LINE_LOOP

Output of Ear clipping

Tessellated parametric surface

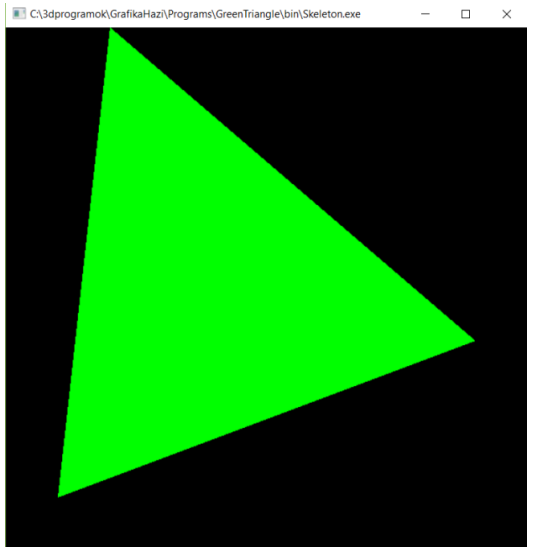
„Convex" polygon



GL_TRIANGLES GL_TRIANGLE_STRIP GL_TRIANGLE_FAN

Graphics hardware and software
Program: Framework and green triangle

Szirmay-Kalos László



OpenGL starters' kit: Shader programs

- `glCreate[Shader|Program]()` creation
- `glShaderSource()` source upload
- `glCompileShader()` compilation
- `glAttachShader()` add shader to program
- `glBindFragDataLocation()` what goes to frame buffer?
- `glLinkProgram()` linking
- `glUseProgram()` select for execution
- `glGetUniformLocation()` uniform variable address
- `glUniform*()` uniform variable set value

```
class GPUProgram {
   bool create(char * vertShader, char * fragShader,
               char * OutputName, char * geomShader = 0);
   void Use();
   void setUniform(…);
};
```

OpenGL starters' kit

Resource generation, binding, uploading

- `glGen[VertexArrays|Buffers|Textures](1, &id);`
- `glBind[VertexArray|Buffer|Texture](id);`
- `glBufferData(GL_ARRAY_BUFFER, …);`
- `glTexImage2D(GL_TEXTURE_2D, …);`

Connect VBO to input registers

- `glEnableVertexAttribArray(reg)` enable register
- `glVertexAttribPointer(reg, …)` buffer to register

Drawing and pipeline management

- `glDrawArrays(type, start, num)` draw call
- `glClearColor(r, g, b, a)` bacground clear color
- `glClear(buffer)` clear background
- `glViewport(vx, vy, vw, vh)` viewport
- `glPointSize(s)` point size
- `glLineWidth(w)` line width

framework.h

```
include: <stdio.h>, <stdlib.h>, <math.h>, <vector>, <string>
         if windows <windows.h>
         <GL/glew.h>, <GL/freeglut.h> // must be downloaded

const unsigned int windowWidth = 600, windowHeight = 600;

struct vec2;
struct vec3;
struct vec4;
struct mat4;

struct Texture {
   unsigned int textureId;
   void create(…);
};

class GPUProgram {
   bool create(char * vertShader,
               char * fragShader, char * OutputName,
               char * geomShader = nullptr);
   void Use();
   void setUniform(…);
};
```

framework.cpp

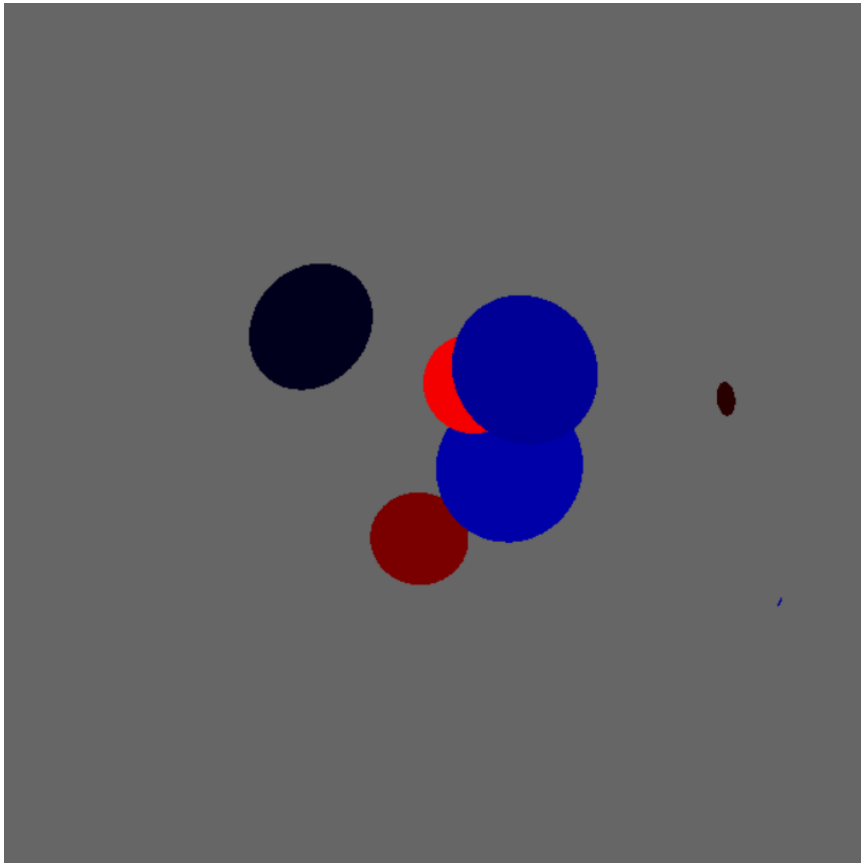
```
#include "framework.h"

void onInitialization(); // Init
void onDisplay(); // Redraw
void onKeyboard(unsigned char key, int pX, int pY); // Key pressed
void onKeyboardUp(unsigned char key, int pX, int pY); // Key released
void onMouseMotion(int pX, int pY); // Move mouse with key pressed
void onMouse(int button, int state, int pX, int pY); // Mouse click
void onIdle(); // Time elapsed

int main(int argc, char * argv[]) {
    glutInit(&argc, argv); glutInitContextVersion(3, 3);
    glutInitWindowSize(windowWidth, windowHeight);
    glutInitWindowPosition(100, 100);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutCreateWindow(argv[0]);

    glewExperimental = true; glewInit(); // no opengl calls before this

    onInitialization();
    glutDisplayFunc(onDisplay); // Register event handlers
    glutMouseFunc(onMouse);
    glutIdleFunc(onIdle);
    glutKeyboardFunc(onKeyboard);
    glutKeyboardUpFunc(onKeyboardUp);
    glutMotionFunc(onMouseMotion);
    glutMainLoop(); return 1;
}
```

Skeleton.cpp

```
#include "framework.h"

const char * const vertexSource;
const char * const fragmentSource;

GPUProgram gpuProgram; // vertex and fragment shaders

void onInitialization() {
    …
    gpuProgram.create(vertexSource, fragmentSource, "outColor");
}

void onDisplay() {
    glClearColor(0, 0, 0, 0);     // background color
    glClear(GL_COLOR_BUFFER_BIT); // clear frame buffer
    …
    glutSwapBuffers(); // exchange buffers for double buffering
}

void onKeyboard(unsigned char key, int pX, int pY) { … }
void onKeyboardUp(unsigned char key, int pX, int pY) { … }
void onMouseMotion(int pX, int pY) { … }
void onMouse(int button, int state, int pX, int pY) { … }
void onIdle() { … }
```

Charges in Soup

Homework 1



Specification

Particles of random charge and mass swim in a fluid of no charge in 2D Euclidean space. Particles are circles in this space, where the radius is proportional to the mass, the intensity is proportional to the absolute value of the charge, the hue is red for positive and green or blue for negative charges. Particles are moving according to the 2D Coulomb force and a soup friction that is proportional to the velocity. Our microscope projects the 2D Euclidean space to 2D hyperbolic space, then transforms it to a unit circle with the Beltrami-Poincaré disc model. This unit circle is shown in the viewport of 600x600 resolution. Pressing the s,d,x,e keys, the virtual world can be translated to left, right, down and up with 0.1 unit. The timestep of the simulation is 0.01 sec even if the onIdle events come slower.

Initially 2 particles exist, by each pressing of the SPACE, a new particle is introduced.

Simulation

$$\frac{d\vec{v}}{dt} = \frac{\sum \vec{F}}{m}$$

$$\frac{d\vec{r}}{dt} = \vec{v}$$

State: $\boldsymbol{r}_i$, $\boldsymbol{v}_i$

```
for(t = 0; t < T; t += dt) { // onIdle
    for each node i {
```

$$\sum \vec{F} = \sum \frac{q_i q_j}{2\pi\varepsilon d_{ji}} \overrightarrow{e_{ji}} - \rho \overrightarrow{\boldsymbol{v}_i}$$

$$\boldsymbol{v}_i \mathrel{+}= \sum \vec{F} / m \cdot dt$$

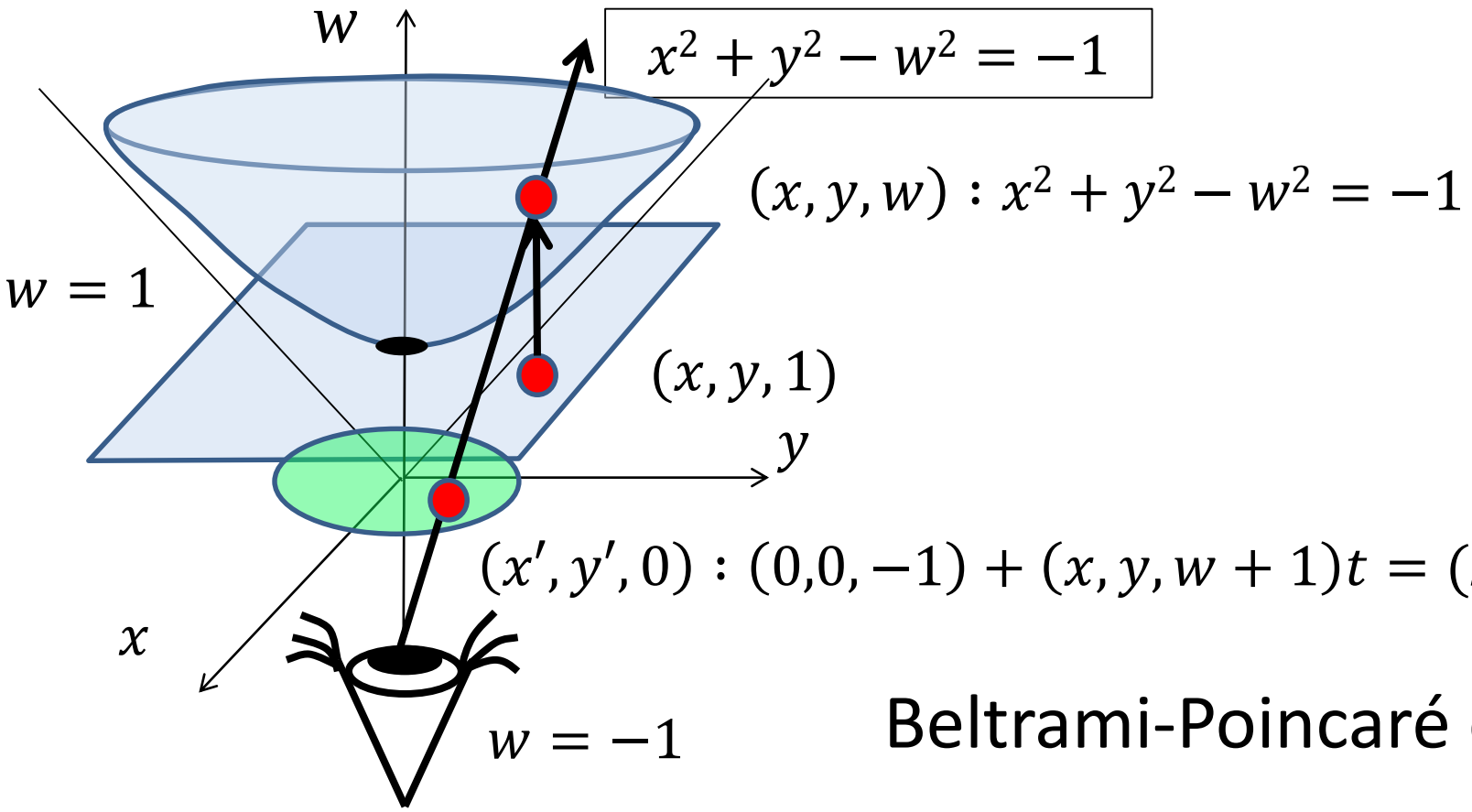
$$\boldsymbol{r}_i \mathrel{+}= \boldsymbol{v}_i \cdot dt$$

```
    }
}
```

View V and Projection P transformation (vertex shader)

V: translation that moves the camera to (0,0,1)

P: maps the plane to a unit circle



Beltrami-Poincaré disc

Steps of solution

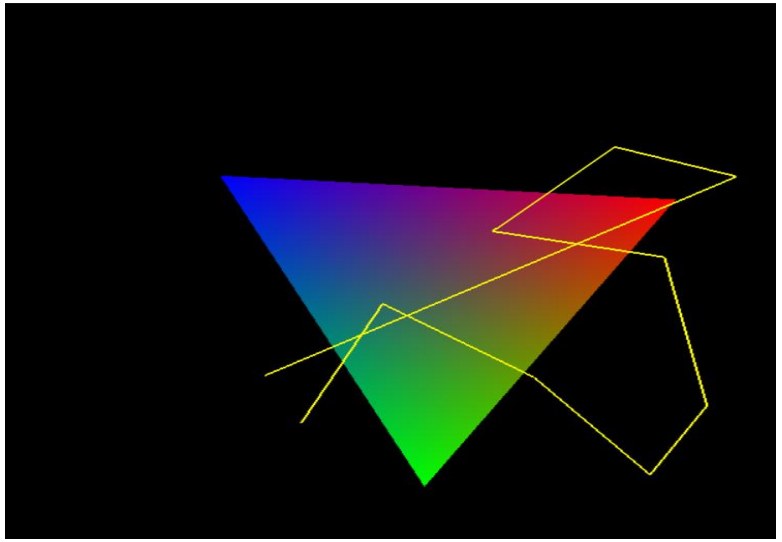
- Atom: transformed circle (triangle-fan)
  - own or shared vao/vbo
  - Properties: mass, charge, (color)
  - State: position, velocity, force
- Virtual world: collection of atoms
- Simulation in onIdle
- Rendering in onDisplay
- World coordinates: Euclidean space
- Vertex shader:
  - Modeling transformation: (scaling +) translation
  - View transformation: translation
  - Projection transformation: hyperbolic/Poincaré
- Pixel shader: constant color

Graphics hardware and software
Program: Vasarely painting

Szirmay-Kalos László

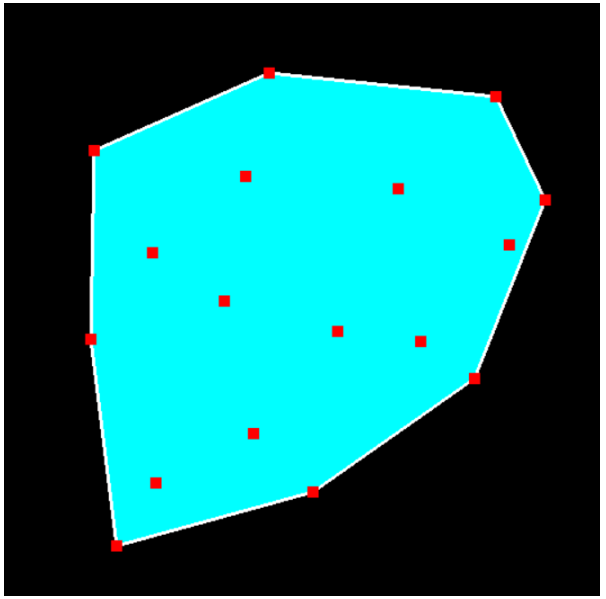
Graphics hardware and software
Program: Animation and interaction

Szirmay-Kalos László



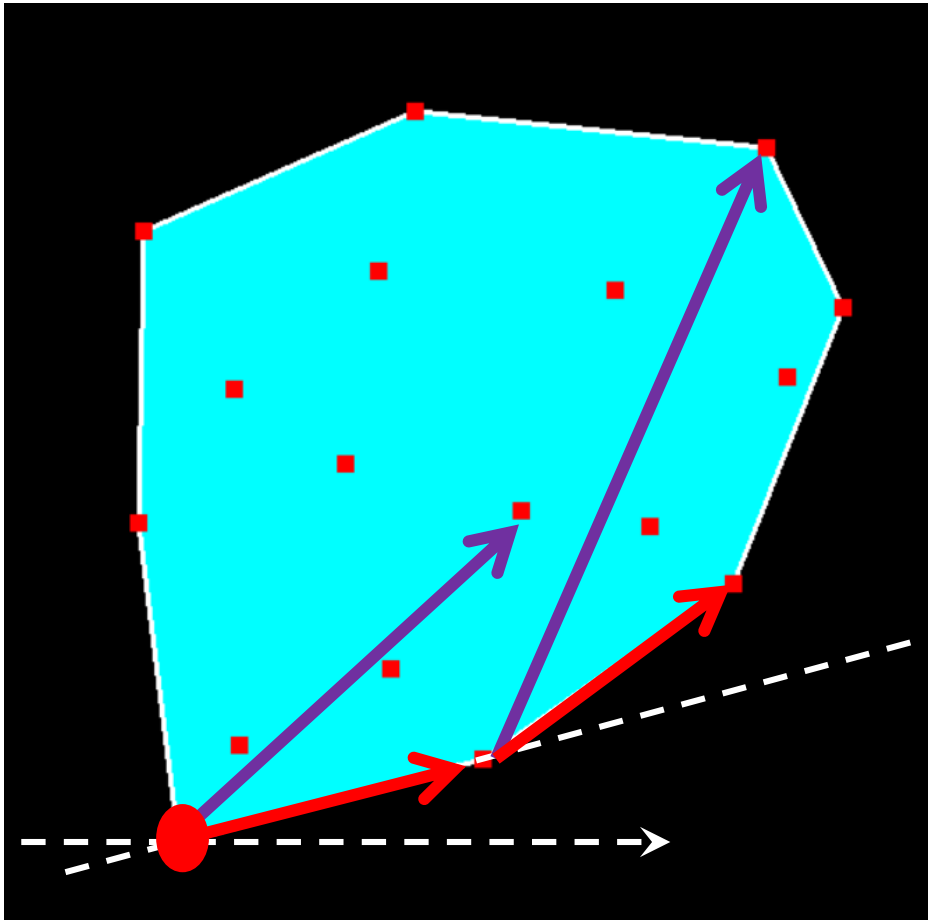
Graphics hardware and software
Program: Convex hull, interaction

Szirmay-Kalos László



Convex hull

Minimal convex set that contains the given points



Start the lowest point, initial direction is to right

While (not back to start) {
    Next point with minimal turn.
}

Vertex and fragment shaders

Vertex shader:

```
layout(location = 0) in vec2 vertexPosition;

void main() {
   gl_Position = vec4(vertexPosition, 0, 1);
}
```

Fragment shader:

```
uniform vec3 color;
out vec4 fragmentColor;

void main() {
   fragmentColor = vec4(color, 1);
}
```

Object

```
struct Object {
   unsigned int vao, vbo; // gpu
   std::vector<vec2> vtx; // cpu

   Object() {
      glGenVertexArrays(1, &vao); glBindVertexArray(vao);
      glGenBuffers(1, &vbo); glBindBuffer(GL_ARRAY_BUFFER, vbo);
      glEnableVertexAttribArray(0);
      glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);
   }
   void updateGPU() {
      glBindVertexArray(vao); glBindBuffer(GL_ARRAY_BUFFER,vbo);
      glBufferData(GL_ARRAY_BUFFER, vtx.size() * sizeof(vec2),
                   &vtx[0], GL_DYNAMIC_DRAW);
   }
   void Draw(int type, vec3 color) {
      if (vertices.size() > 0) {
         glBindVertexArray(vao);
         gpuProgram.setUniform(color, "color");
         glDrawArrays(type, 0, vertices.size());
      }
   }
};
```

Convex hull

```
class ConvexHull {
   Object p, h; // points and hull
public:
   void addPoint(vec2 pp) { p.vtx.push_back(pp); }
   void update() {
      if (p.vtx.size() >= 3) findHull();
      p.updateGPU();
      h.updateGPU();
   }
   vec2 * pickPoint(vec2 pp) {
      for (auto& v : p.vtx) if (length(pp-v) < 0.05f) return &v;
      return nullptr;
   }
   void findHull();
   void Draw() {
      h.Draw(GL_TRIANGLE_FAN, vec3(0, 1, 1));
      h.Draw(GL_LINE_LOOP, vec3(1, 1, 1));
      p.Draw(GL_POINTS, vec3(1, 0, 0));
   }
};
```

Convex hull generation

```
void ConvexHull::findHull() {
    h.vtx.clear();
    vec2 * vStart = &p.vtx[0]; // Find lowest point
    for (auto& v : p.vtx) if (v.y < vStart->y) vStart = &v;

    vec2 vCur = *vStart, dir(1, 0), *vNext;
    do { // find convex hull points one by one
        float maxCos = -1;
        for (auto& v : p.vtx) { // find minimal left turn
            float len = length(v - vCur);
            if (len > 0) {
                float cosPhi = dot(dir, v - vCur) / len;
                if (cosPhi > maxCos) { maxCos = cosPhi; vNext = &v;}
            }
        }
        h.vtx.push_back(*vNext); // save as convex hull
        dir = normalize(*vNext - vCur); // prepare for next
        vCur = *vNext;
    } while (vStart != vNext);
}
```

Virtual world, initialization and display

```
ConvexHull * hull;
vec2 * pickedPoint = nullptr;

void onInitialization() {
    glViewport(0, 0, windowWidth, windowHeight);
    glLineWidth(2);
    glPointSize(10);
    hull = new ConvexHull;
    gpuProgram.create(vertexSrc, fragmentSrc, "fragmentColor");
}

void onDisplay() {
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    hull->Draw();
    glutSwapBuffers();
}
```

Controller

```
vec2 PixelToNDC(int pX, int pY) { // if full viewport
   return vec2(2.0f * pX / windowWidth - 1; // flip y axis
               1.0f - 2.0f * pY / windowHeight);
}

void onMouse(int button, int state, int pX, int pY) {
   if (button==GLUT_LEFT_BUTTON && state==GLUT_DOWN) {
      hull->addPoint(PixelToNDC(pX, pY));
      hull->update(); glutPostRedisplay(); // redraw
   }
   if (button==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
      pickedPoint = hull->pickPoint(PixelToNDC(pX, pY));
   if (button==GLUT_RIGHT_BUTTON && state==GLUT_UP)
      pickedPoint = nullptr;
}

void onMouseMotion(int pX, int pY) {
   if (pickedPoint) {
      *pickedPoint = vec2(PixelToNDC(pX, pY));
       hull->update(); glutPostRedisplay(); // redraw
   }
}
```

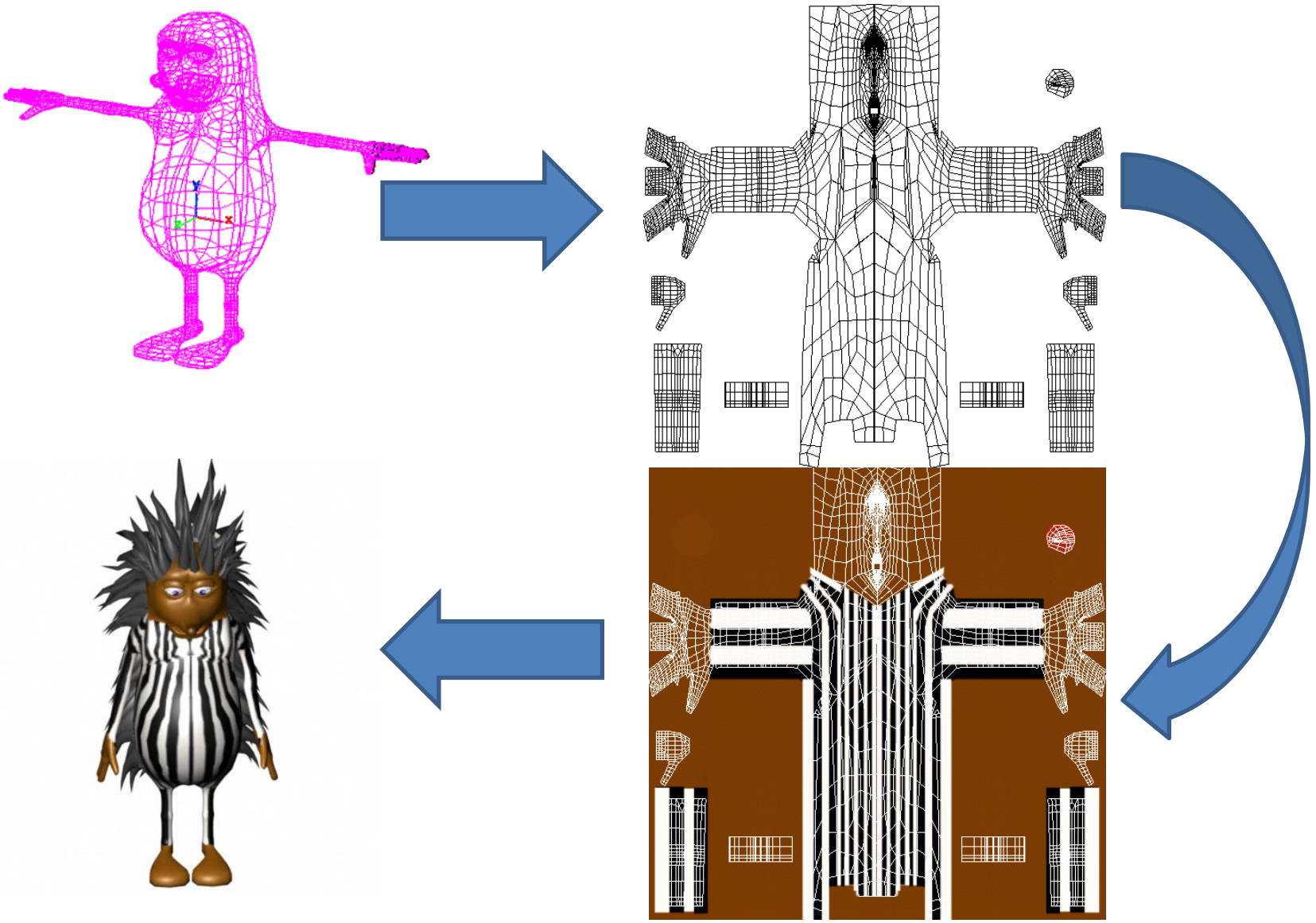
“Everything must be made as simple as possible. But not simpler.”

Albert Einstein

2D texturing

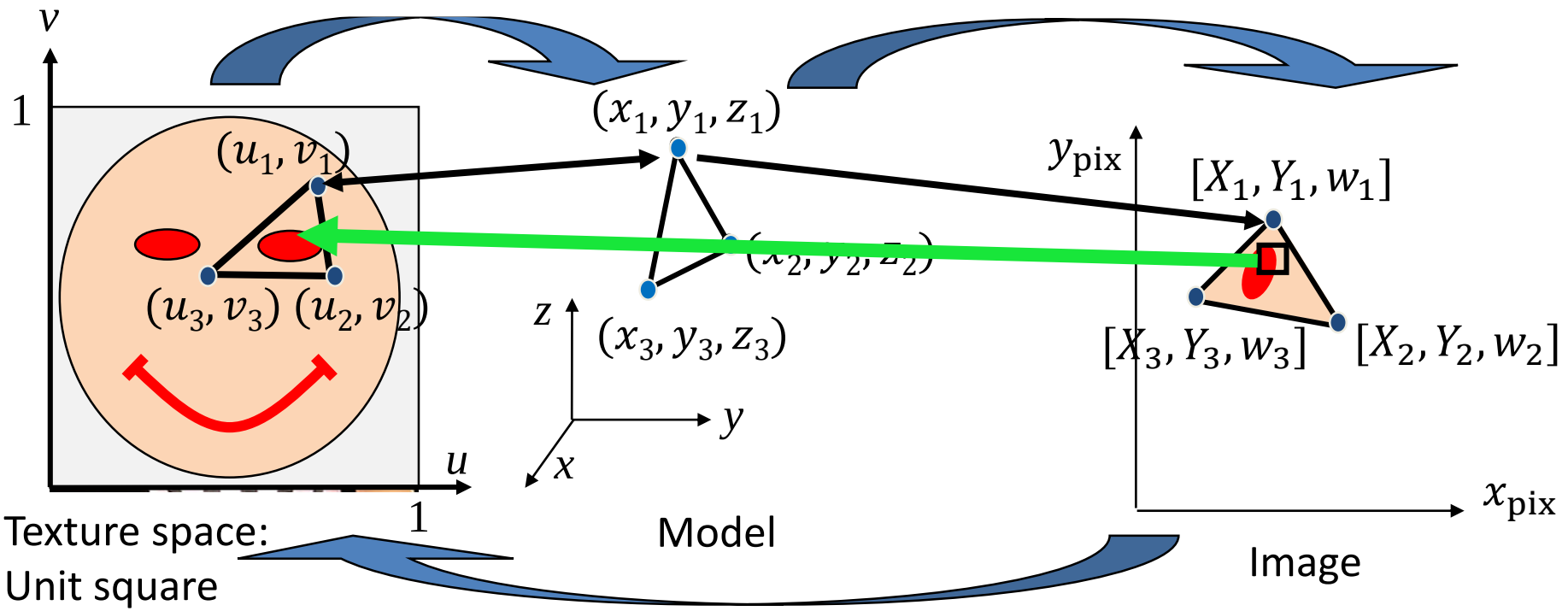
Szirmay-Kalos László

2D texturing



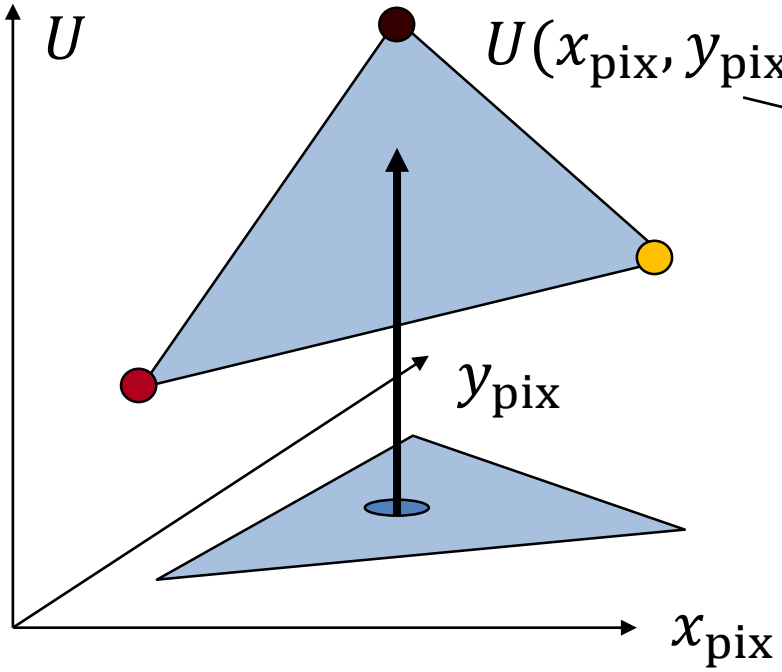
2D texturing

$$[X, Y, w] = [u, v, 1] \cdot \boldsymbol{P}$$
$$(x_{\text{pix}}, y_{\text{pix}}) = [X/w, Y/w]$$

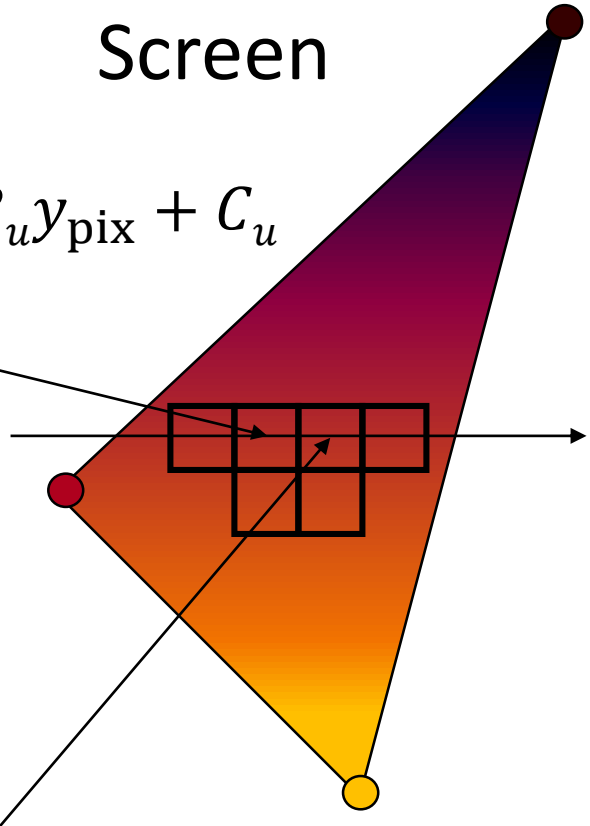


$$[u/w, v/w, 1/w] = [x_{\text{pix}}, y_{\text{pix}}, 1] \cdot \boldsymbol{P}^{-1}$$
$$[U, \quad V, \quad W] \rightarrow u = U/W, \quad v = V/W$$

Interpolation



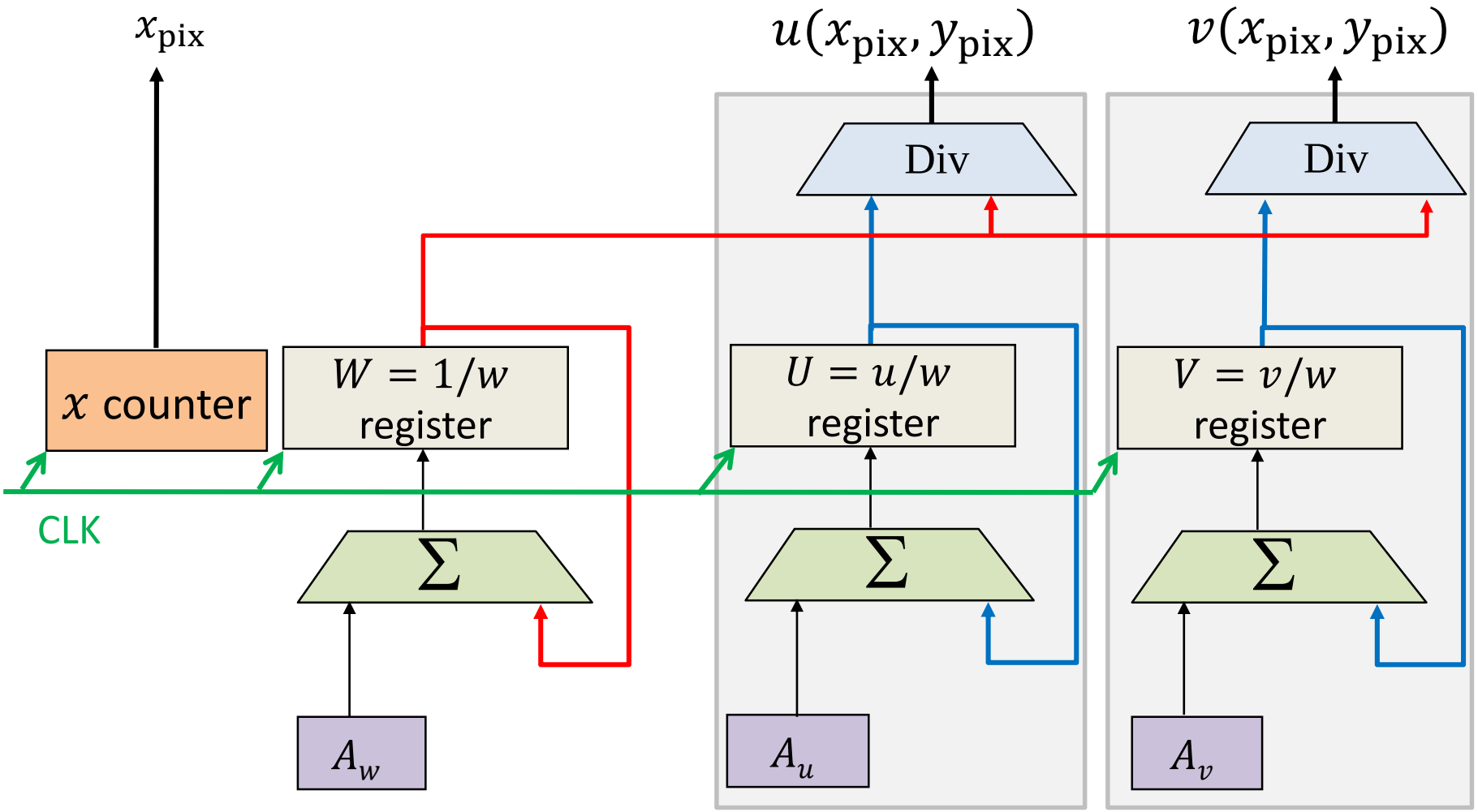
Screen



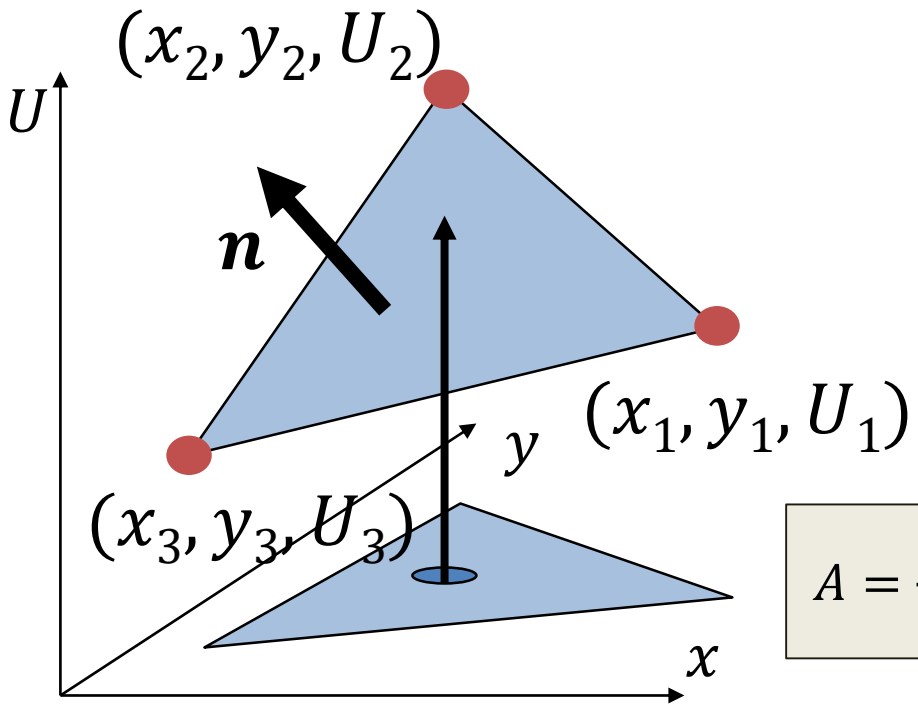
$$U(x_{\text{pix}}, y_{\text{pix}}) = A_u x_{\text{pix}} + B_u y_{\text{pix}} + C_u$$

$$U(x_{\text{pix}} + 1, y_{\text{pix}}) = U(x_{\text{pix}}, y_{\text{pix}}) + A_u$$

Interpolation hardware



Triangle setup



$$U_1 = Ax_1 + By_1 + C$$
$$U_2 = Ax_2 + By_2 + C$$
$$U_3 = Ax_3 + By_3 + C$$

$$A = \frac{(U_3 - U_1)(Y_2 - Y_1) - (Y_3 - Y_1)(U_2 - U_1)}{(X_3 - X_1)(Y_2 - Y_1) - (Y_3 - Y_1)(X_2 - X_1)}$$

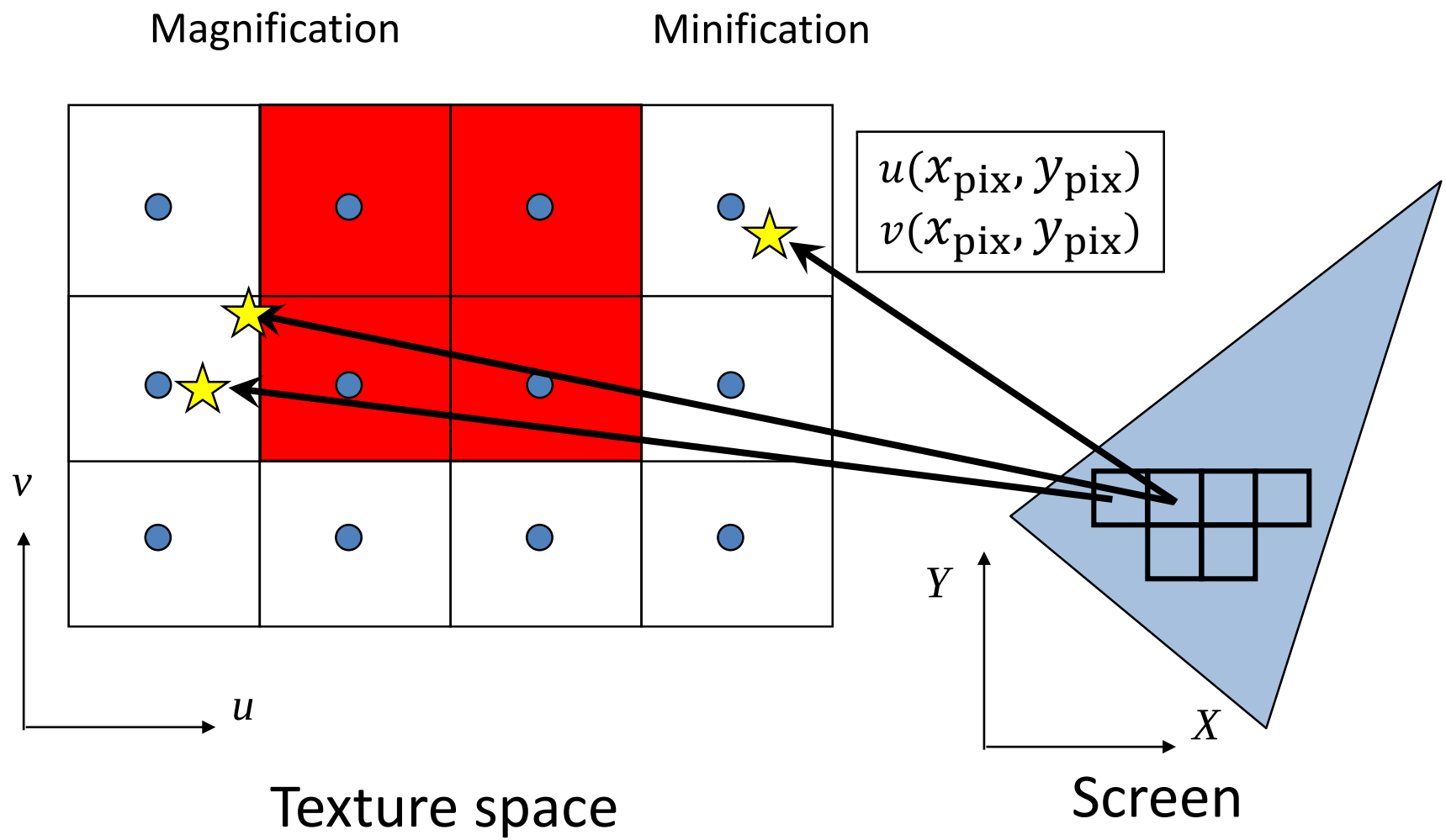
n_x

$-n_u$

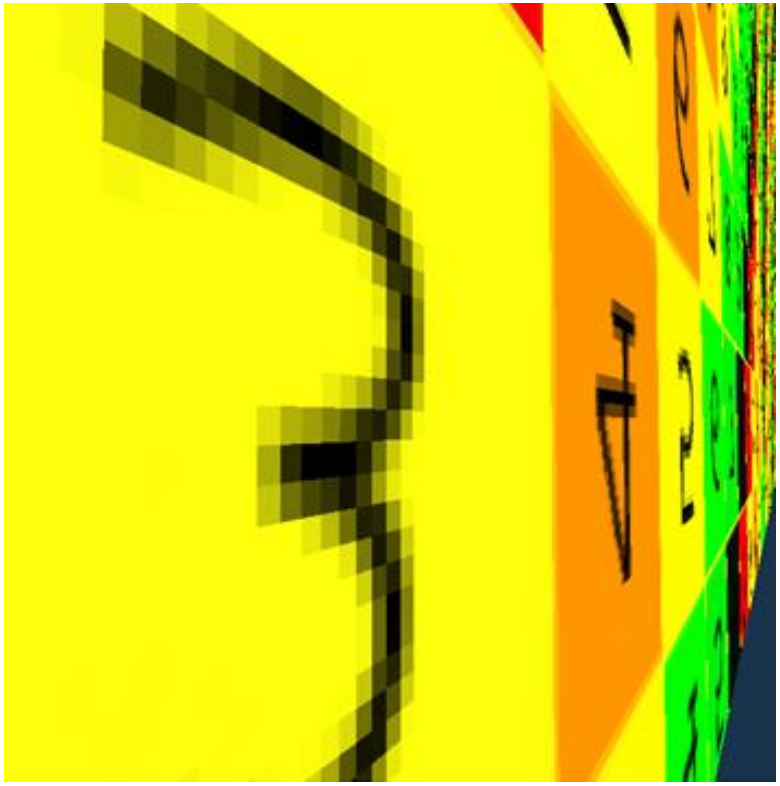
$$U(x, y) = Ax + By + C$$
$$n_x x + n_y y + n_u U + d = 0$$

$$\boldsymbol{n} = (X_2 - X_1, Y_2 - Y_1, U_2 - U_1) \times (X_3 - X_1, Y_3 - Y_1, U_3 - U_1)$$

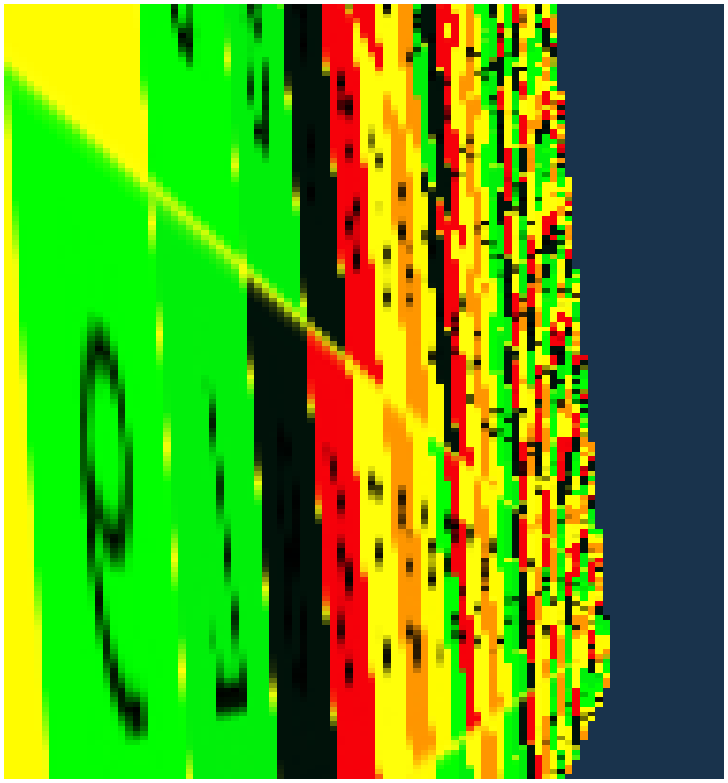
Texture filtering (GL_NEAREST)



Correspondence of texture and image spaces



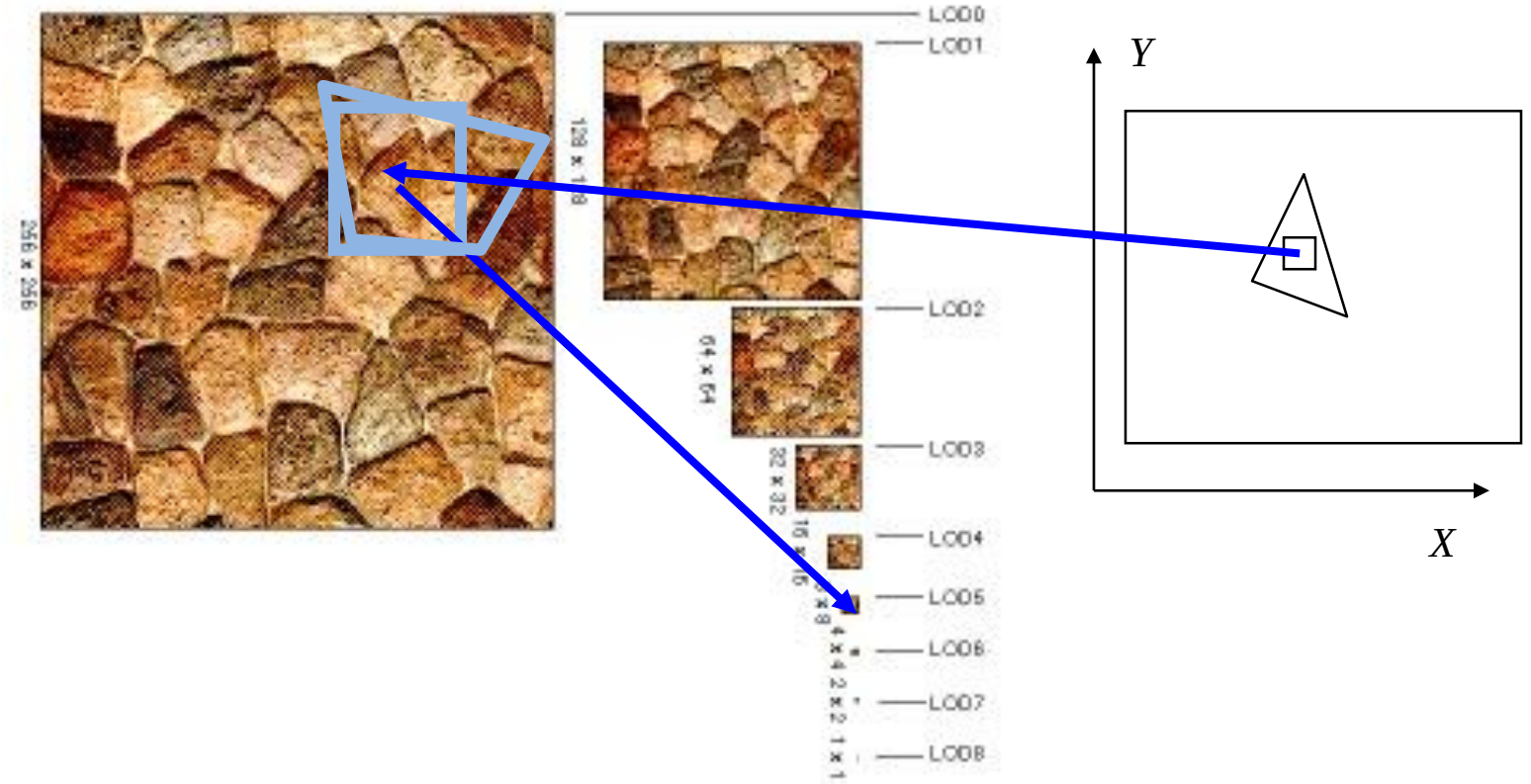
Magnification



Minification

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
```

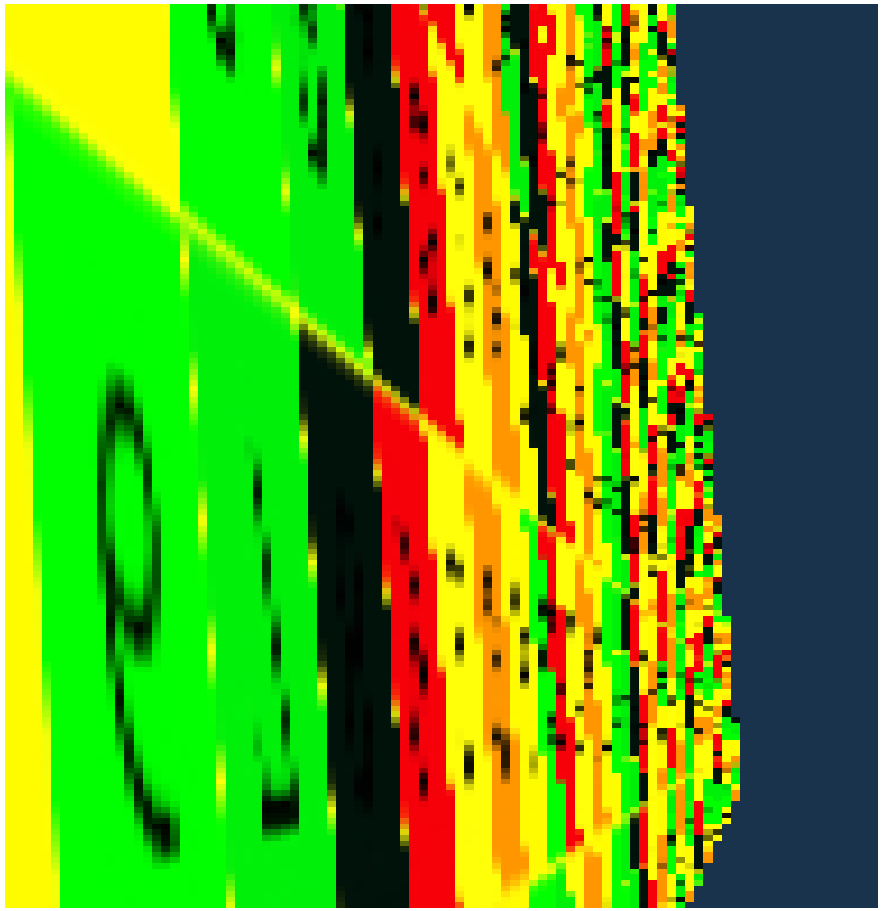
Mip-map (multum in parvo)
Good for minification



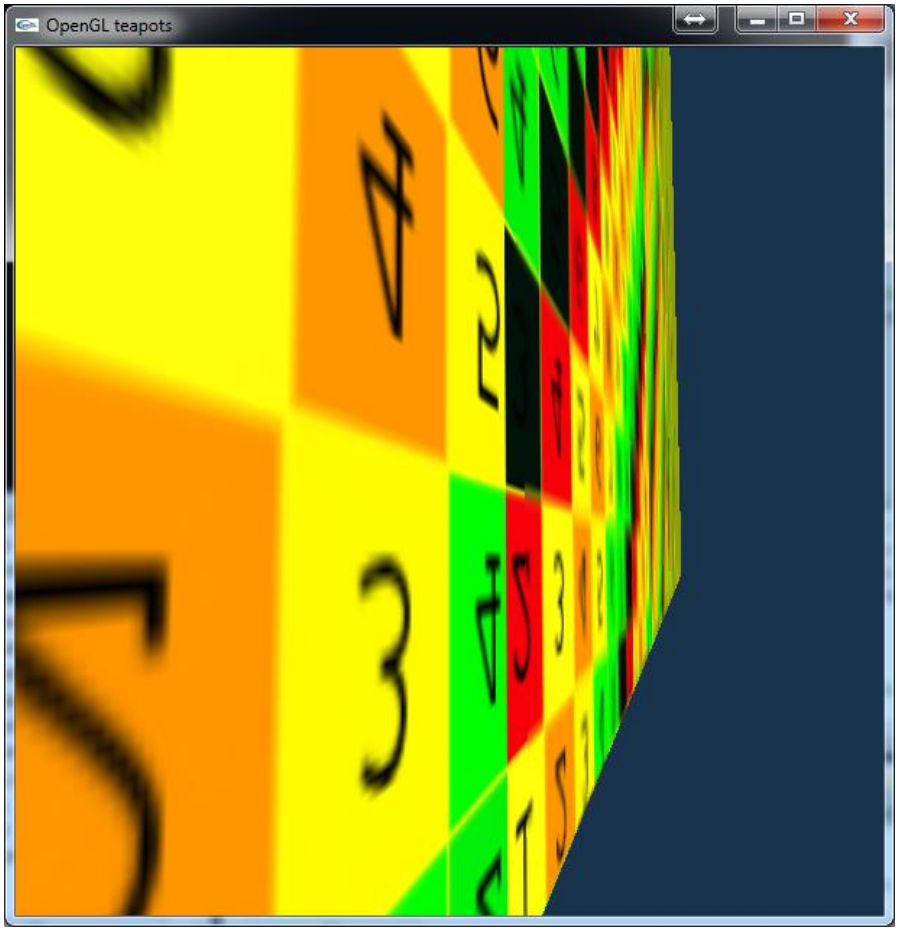
a) `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST); // Mip-mapping`

b) `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR); // Tri-linear filtering`

Mip-map (GL_LINEAR_MIPMAP_...)

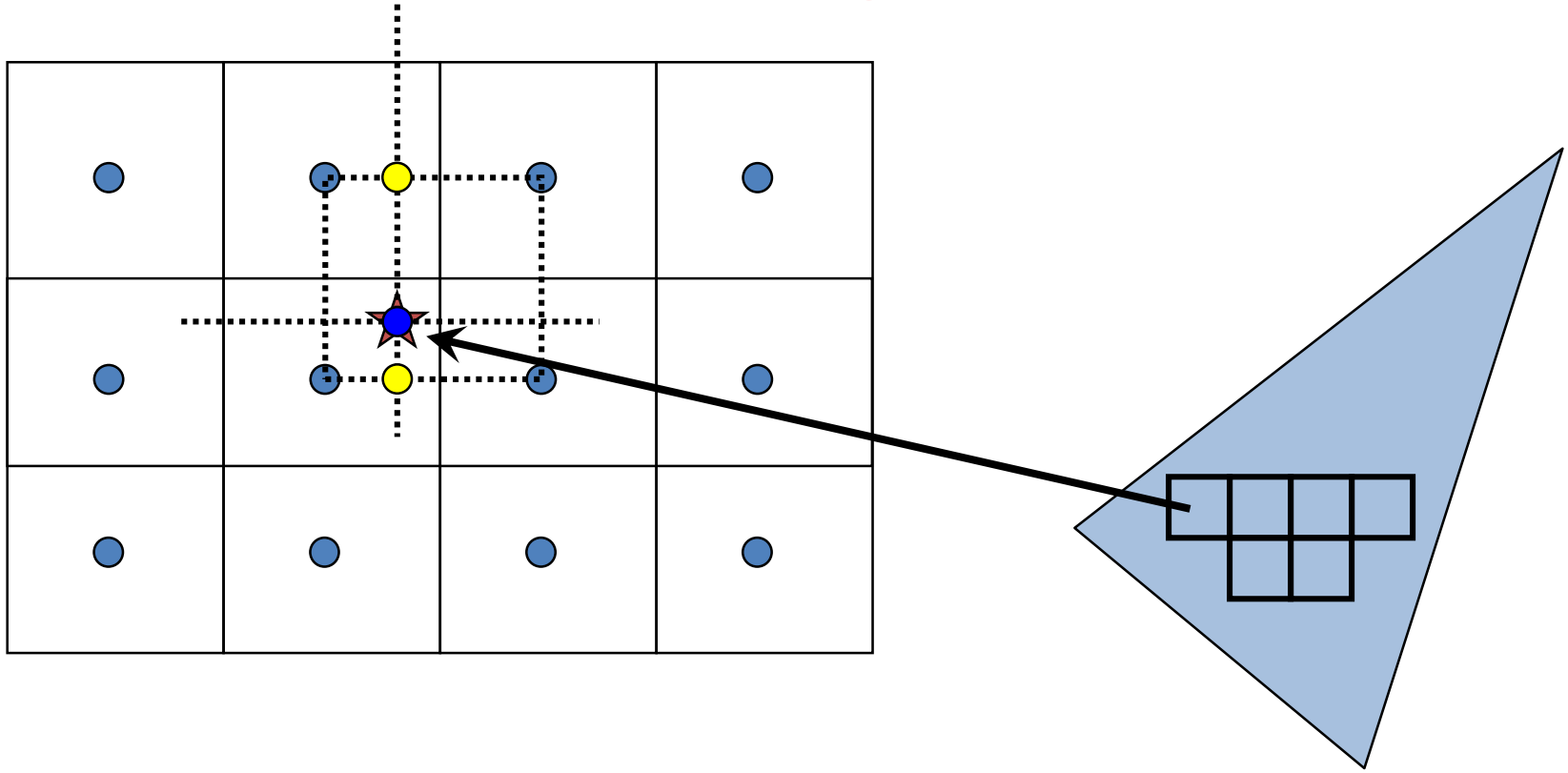


`GL_NEAREST`



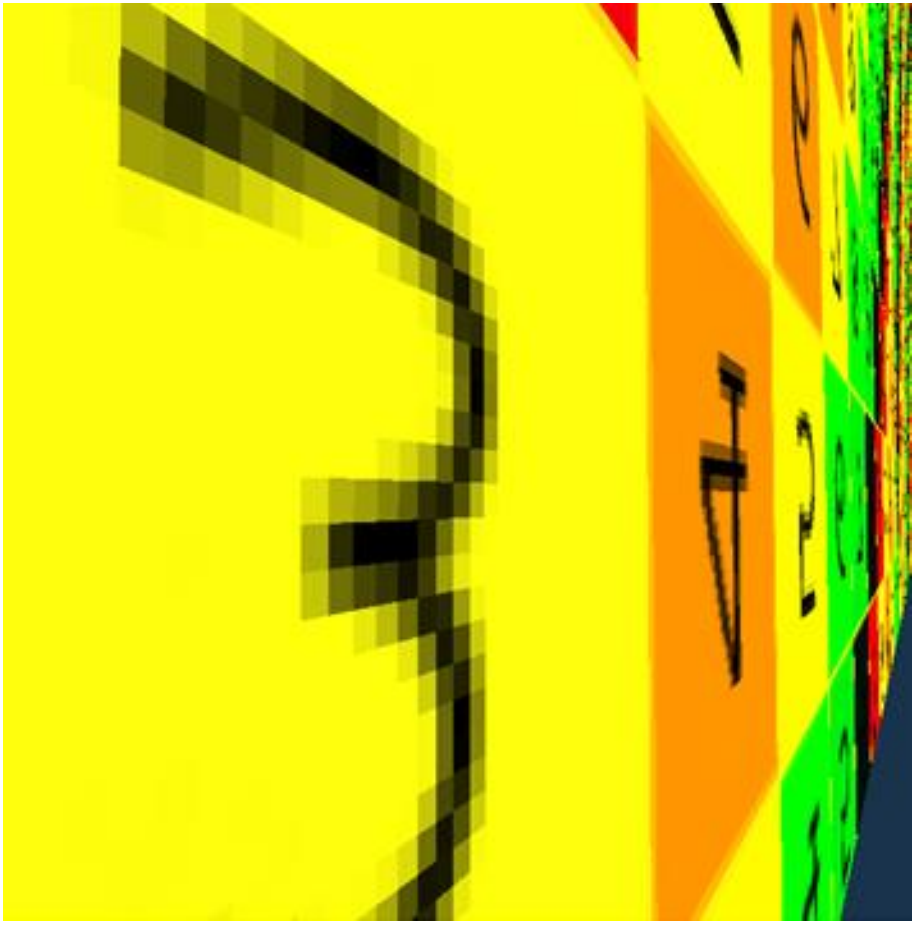
`GL_LINEAR_MIPMAP_NEAREST`

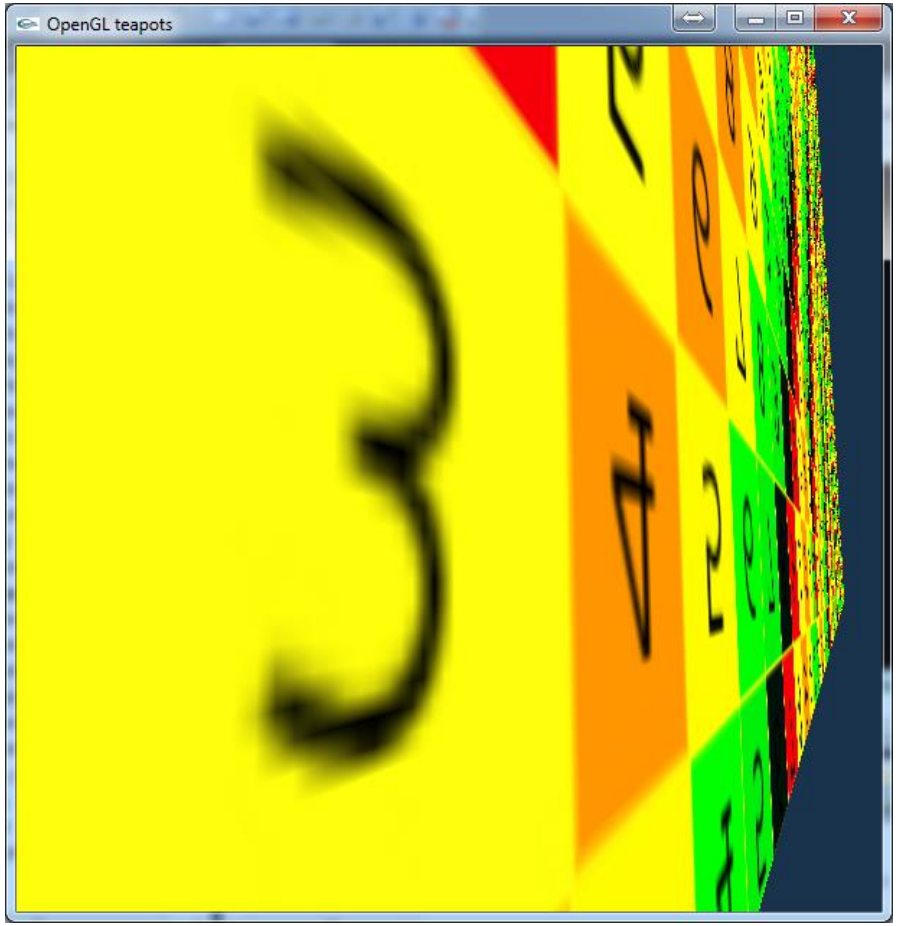
Bi-linear texture filtering (GL_LINEAR)
Good for magnification



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

Bi-linear filtering (GL_LINEAR)

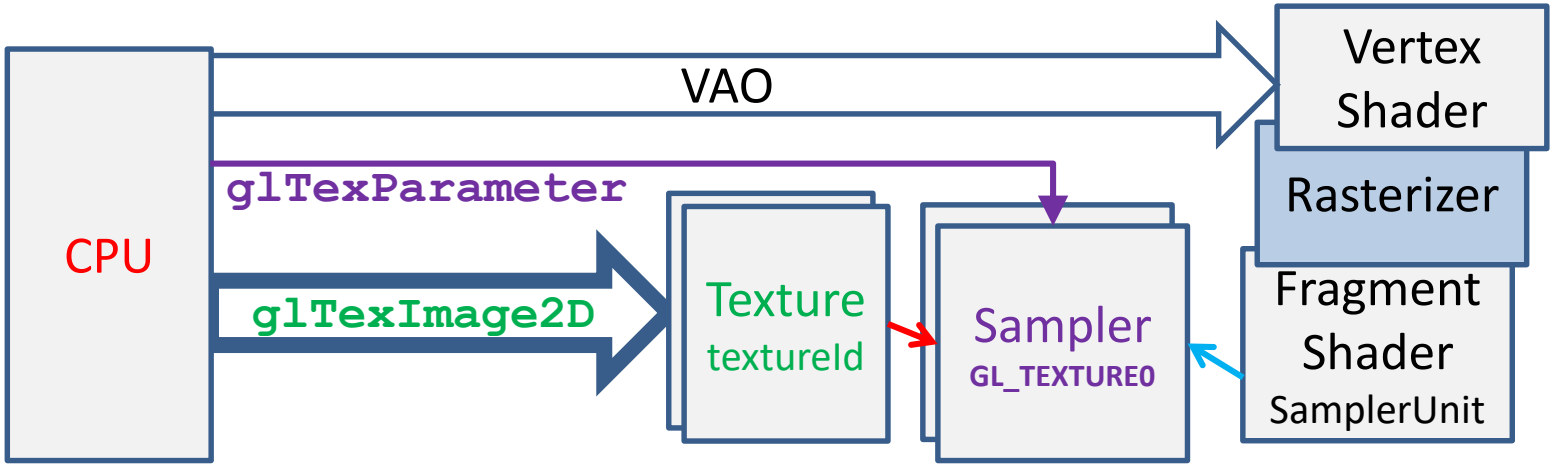




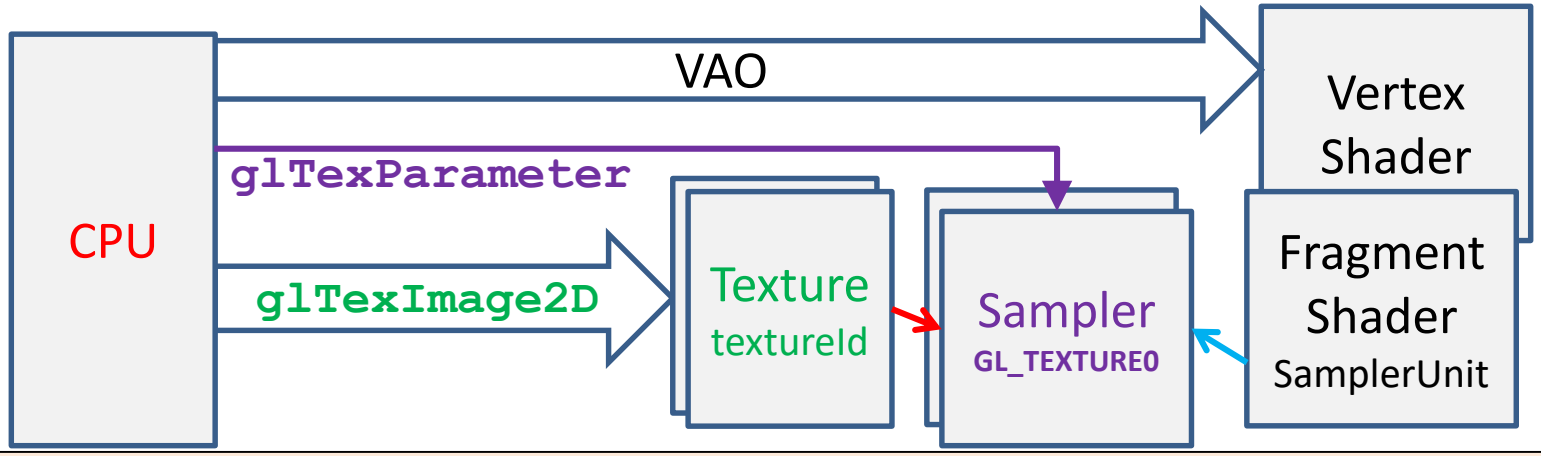
GL_NEAREST

GL_LINEAR

Texturing support of the GPU



Texturing 1: Upload to GPU




```
unsigned int textureId;

void UploadTexture(int width, int height, vector<vec4>& image) {
  glGenTextures(1, &textureId);
  glBindTexture(GL_TEXTURE_2D, textureId);      // binding

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
               GL_RGBA, GL_FLOAT, &image[0]); //Texture -> GPU

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}
```

Your responsibility

Mip-map level

Border

Texturing 2: Equip objects with texture coordinates

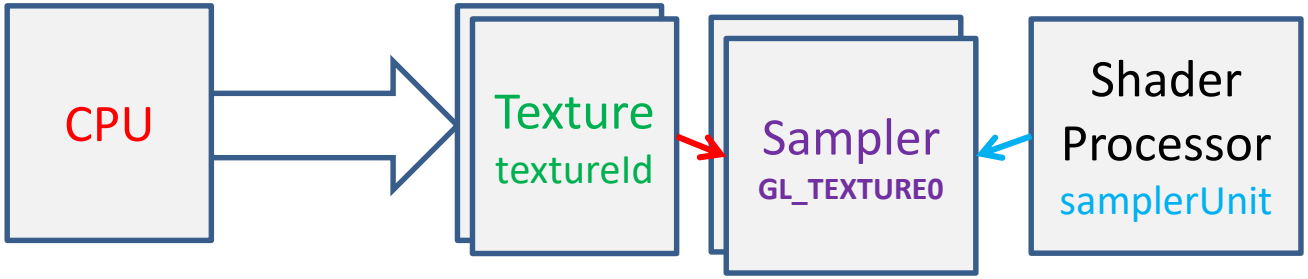
```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(2, vbo);// Generate 2 vertex buffer objects

// vertex coordinates: vbo[0] -> Attrib Array 0 -> vertices
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
float vtxs[] = {x1, y1, x2, y2, …};
glBufferData(GL_ARRAY_BUFFER, sizeof(vtxs),vtxs, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);

// vertex coordinates: vbo[1] -> Attrib Array 1 -> uvs
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
float uvs[] = {u1, v1, u2, v2, …};
glBufferData(GL_ARRAY_BUFFER, sizeof(uvs), uvs, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, NULL);
```

Texturing 3: Vertex és Pixel Shader

```
layout(location = 0) in vec2 vtxPos;
layout(location = 1) in vec2 vtxUV;

out vec2 texcoord;

void main() {
    gl_Position = vec4(vtxPos, 0, 1) * MVP;
    texcoord = vtxUV;
    …
}
```

```
uniform sampler2D samplerUnit;
in vec2 texcoord;
out vec4 fragmentColor;

void main() {
   fragmentColor = texture(samplerUnit, texcoord);
}
```

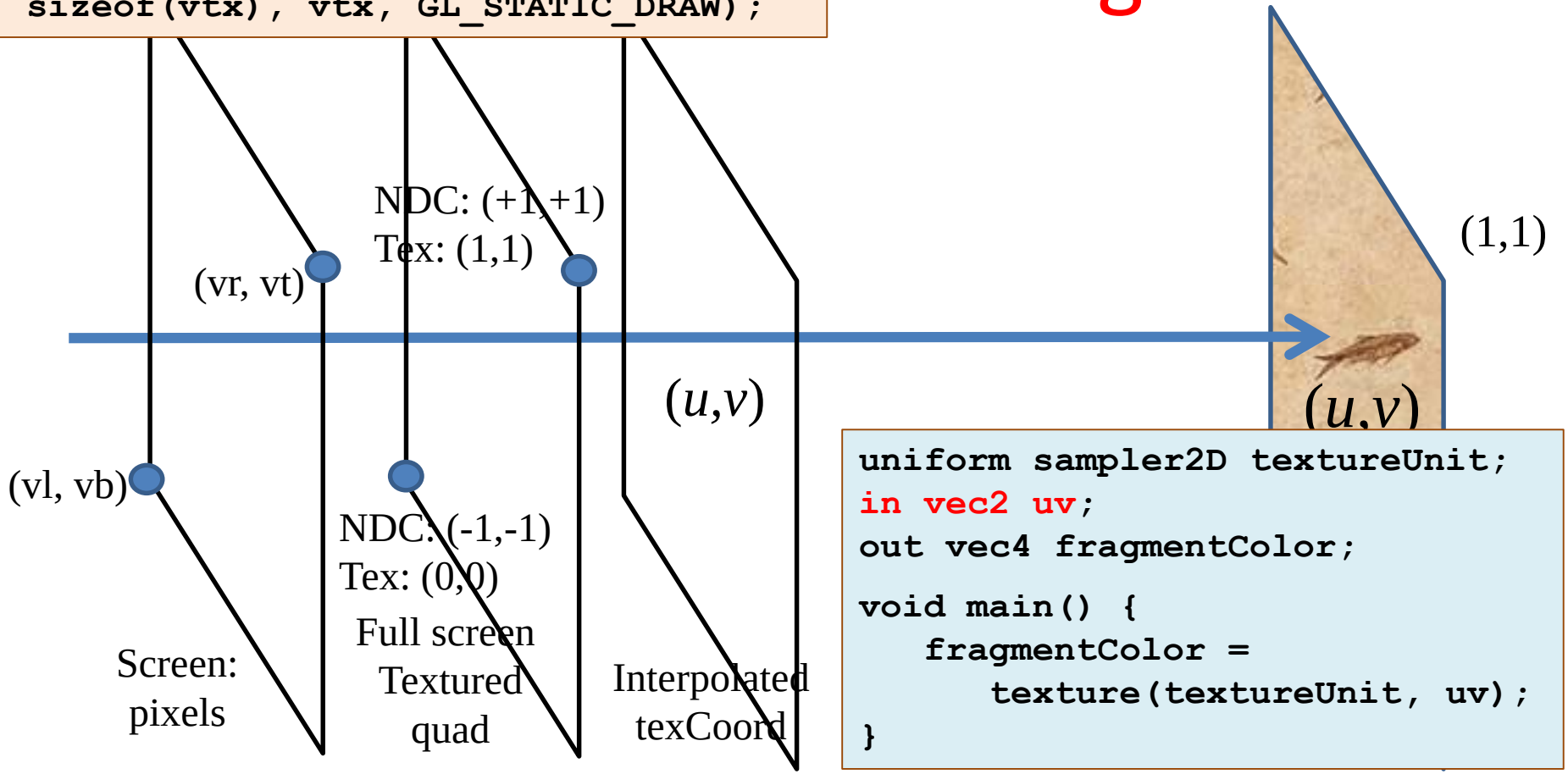
Texturing 4: Active texture and sampler




```
unsigned int textureId;

void Draw( ) {
  int sampler = 0; // which sampler unit should be used

  int location = glGetUniformLocation(shaderProg, "samplerUnit");
  glUniform1i(location, sampler);

  glActiveTexture(GL_TEXTURE0 + sampler);  // = GL_TEXTURE0
  glBindTexture(GL_TEXTURE_2D, textureId);

  glBindVertexArray(vao);
  glDrawArrays(GL_TRIANGLES, 0, nVtx);
}
```

```
float vtx = {-1,-1, 1,-1, 1,1, -1, 1};
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vtx), vtx, GL_STATIC_DRAW);
```

Image viewer

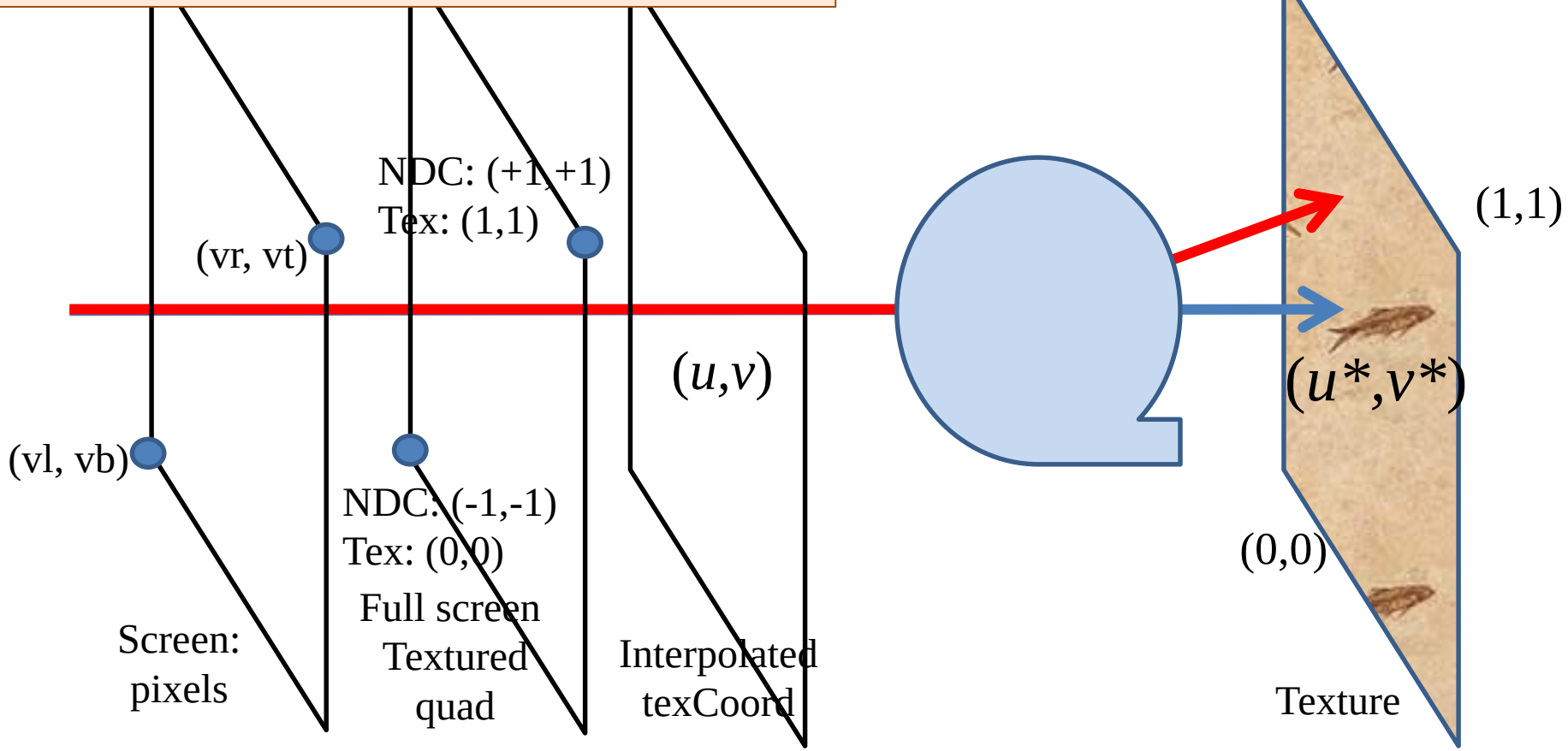



```
uniform sampler2D textureUnit;
in vec2 uv;
out vec4 fragmentColor;

void main() {
   fragmentColor =
      texture(textureUnit, uv);
}
```

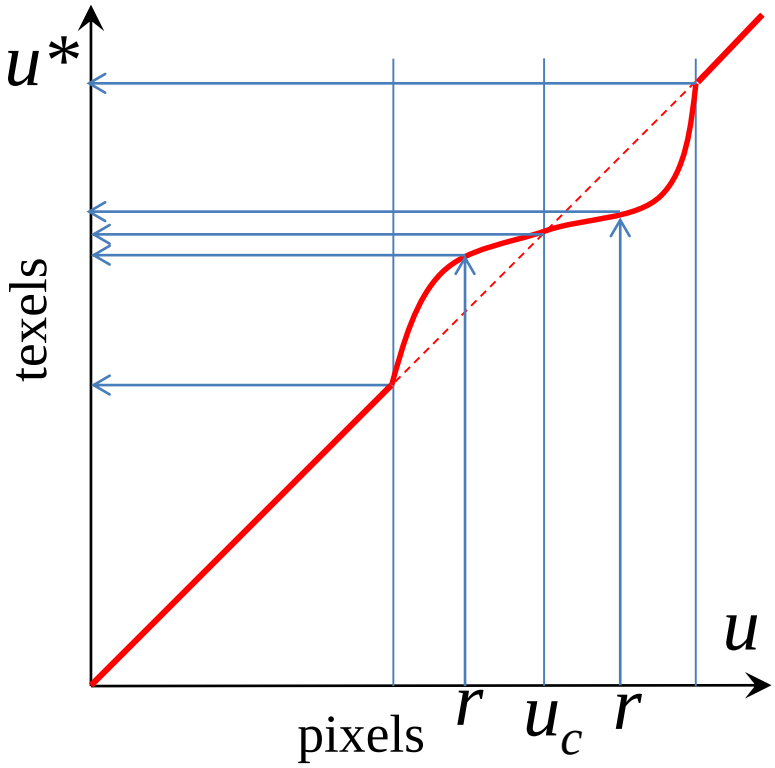
```
layout(location = 0) in vec2 vp; // Attrib Array 0
out vec2 uv;// output attribute

void main() {
   uv = (vp + vec2(1, 1)) / 2; // clipping to texture space
   gl_Position = vec4(vp.x, vp.y, 0, 1);
}
```

```
float vtx = {-1,-1, 1,-1, 1,1, -1, 1};
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vtx), vtx, GL_STATIC_DRAW);
```

Image viewer




```
layout(location = 0) in vec2 vp; // Attrib Array 0
out vec2 uv;// output attribute

void main() {
   uv = (vp + vec2(1, 1)) / 2; // clipping to texture space
   gl_Position = vec4(vp.x, vp.y, 0, 1);
}
```



Magic lense

$$u^* = \frac{(u - u_c)^3}{r^2} + u_c \quad \text{if } |u - u_c| < r$$

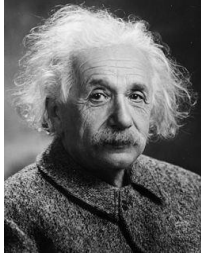
```
uniform sampler2D textureUnit;
uniform vec2 uvc; // cursor position in texture space

in vec2 uv;      // interpolated texture coordinates
out vec4 fragmentColor;

void main() {
   const float r2 = 0.05f;
   float d2 = dot(uv - uvc, uv - uvc);
   vec2 tuv = (d2 < r2) ? (uv - uvc) * d2 / r2 + uvc : uv;
   fragmentColor = texture(textureUnit, tuv);
}
```

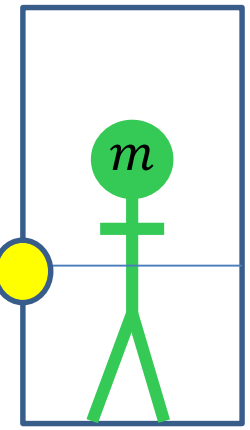
Swirl


```
uniform sampler2D textureUnit;
uniform vec2 uvc; // cursor position in texture space

in vec2 uv;     // interpolated texture coordinates
out vec4 fragmentColor;

void main() {
   const float a = 8, alpha = 15;
   float ang = a * exp( -alpha * length(uv - uvc) );
   mat2 rotMat = mat2( cos(ang), sin(ang),
                      -sin(ang), cos(ang) );

   vec2 tuv = (uv - uvc) * rotMat + uvc;
   fragmentColor = texture(textureUnit, tuv);
}
```

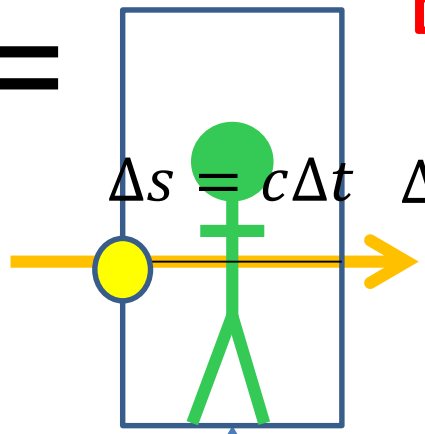
Gravity (black hole)

Equivalence principle





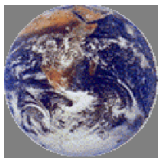
=



$\Delta s = c\Delta t$

$$\Delta d = \frac{g}{2}(\Delta t)^2 = \frac{fM}{2r^2c^2}(\Delta s)^2 = \frac{r_0}{4r^2}(\Delta s)^2$$

$\frac{r_0}{2}$: Schwarzschild radius

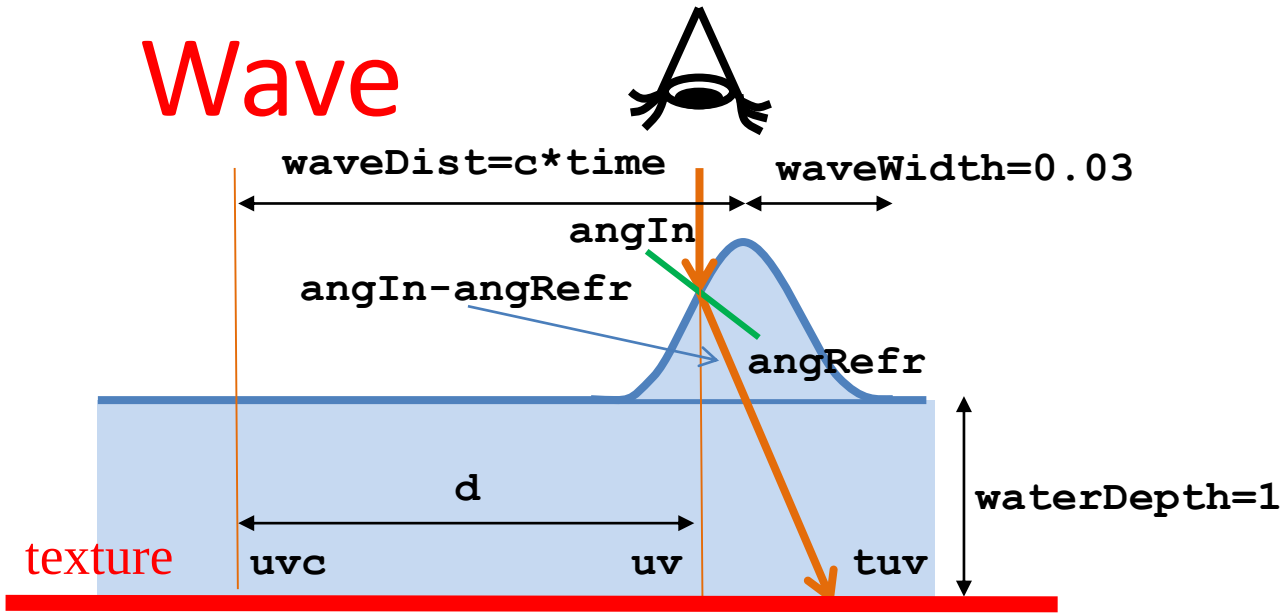


$$g = \frac{fM}{r^2}$$

g


```
void main() {
    const float r0 = 0.09f, ds = 0.001f;
    vec3 p = vec3(uv,0), dir = vec3(0,0,1), blackhole = vec3(uvc,0.5f);
    float r2 = dot(blackhole - p, blackhole - p);
    while (p.z < 1 && r2 > r0 * r0) {
        p += dir * ds;
        r2 = dot(blackhole - p, blackhole - p);
        vec3 gDir = (blackhole - p)/sqrt(r2); // gravity direction
        dir = normalize(dir * ds + gDir * r0 / r2 / 4 * ds * ds);
    }
    if (p.z >= 1) fragmentColor = texture(textureUnit,vec2(p.x,p.y));
    else          fragmentColor = vec4(0, 0, 0, 1);
}
```

Wave




```
uniform float time;
const float PI = 3.14159265, n = 1.33, c = 0.1, aMax = 0.1;

void main() {
   float d = length(uv - uvc), waveDist = c * time;
   if (abs(d - waveDist) < waveWidth) {
      float angIn = aMax/waveDist * sin((waveDist-d)/waveWidth*PI);
      float angRefr = asin(sin(angIn)/n);
      vec2 dir = (uv - uvc)/d;
      vec2 tuv = uv + dir * tan(angIn - angRefr) * waterDepth;
      fragmentColor = texture(textureUnit, tuv);
   } else {
      fragmentColor = texture(textureUnit, uv);
   }
}
```