OpenGL

Dr Tomasz Bednarz

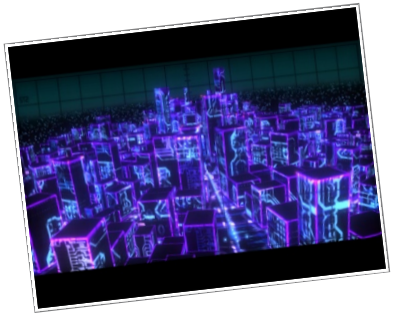
Agenda for today

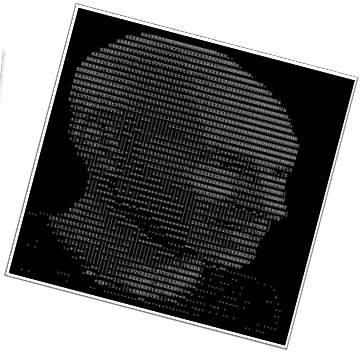
- Introduction and motivation
- Briefly on OpenGL + OpenCL and Compute Shaders
- Vulkan API
- OpenGL shading language
  - Colour
  - GLSL and Shaders
- Transformations
- Lighting, Shading and Texture Mapping
- Geometry and Tessellation Shaders (Daniel Filonik)
- WebGL (Xavier Ho)

Start with pixels

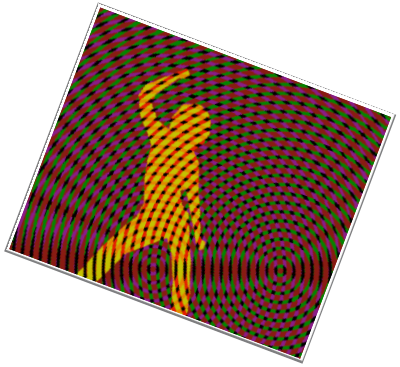
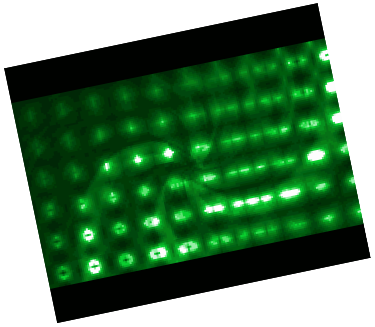
click here

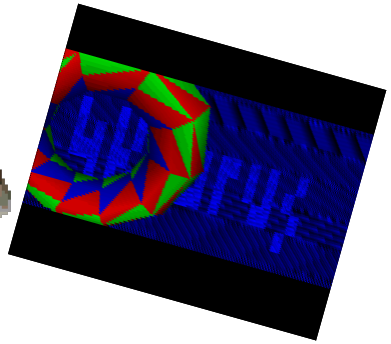
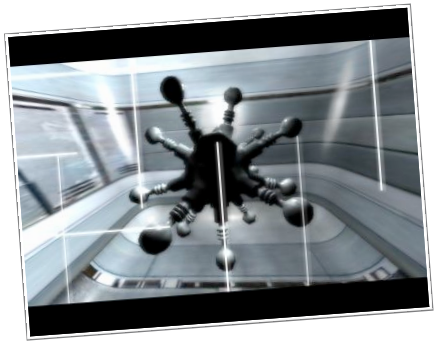
demoscene



demoscene was born in the computer underground, and demos are the product of extreme programming and self-expression

challenging: simple ALU, no FPU or GPU

demo

real-time procedural visualisations (not pre-rendered animations!)

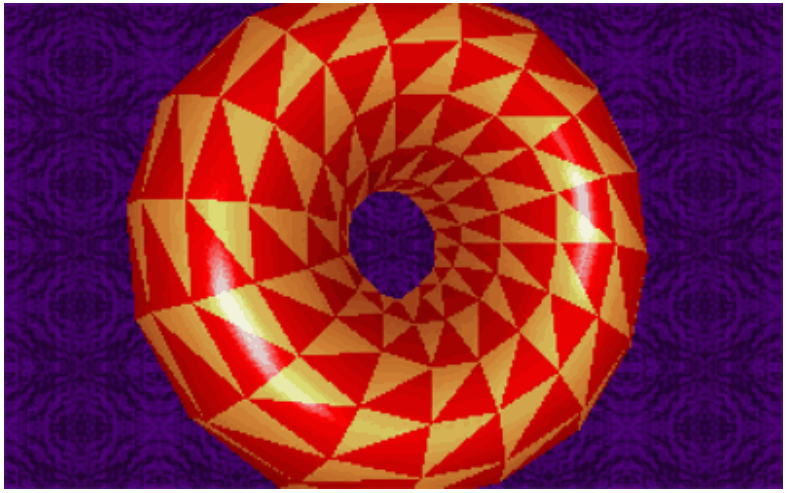
executable

limited in size: 256b, 512b, 1k, 4k, 64k, floppy+

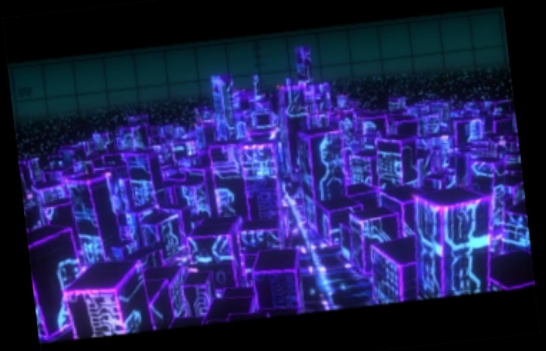
platforms: atari, c64, amiga, pc, web

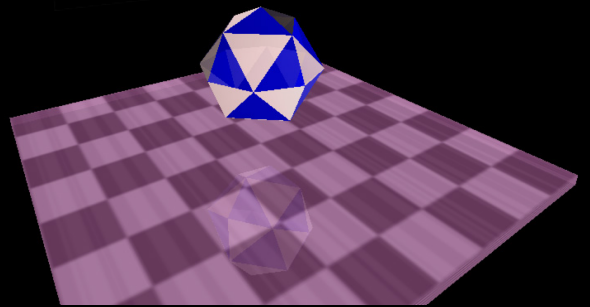
design process:

- group gathering
- music before demo
- demo before music
- code tricks to achieve impossible



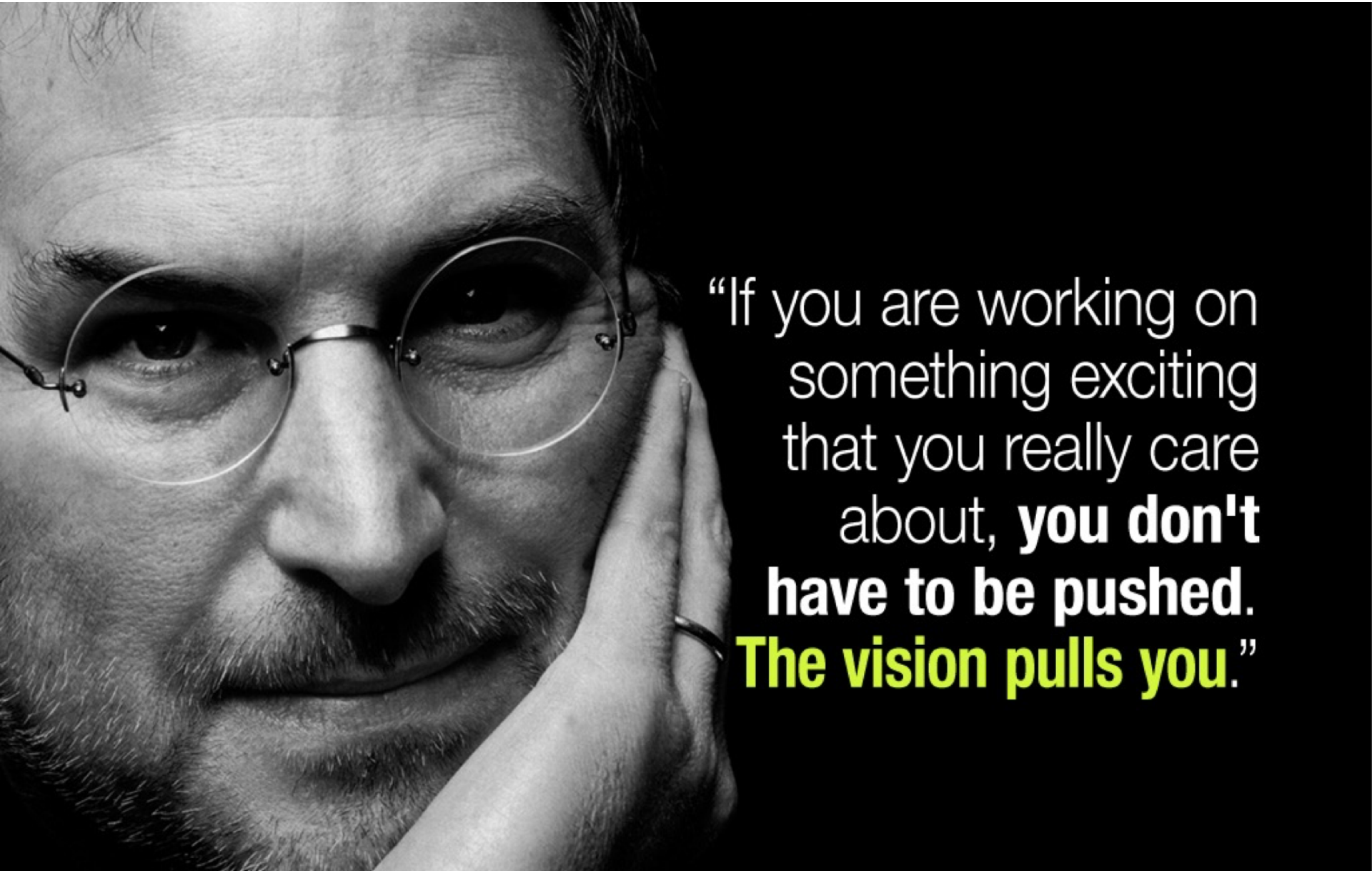
in Europe everybody C++





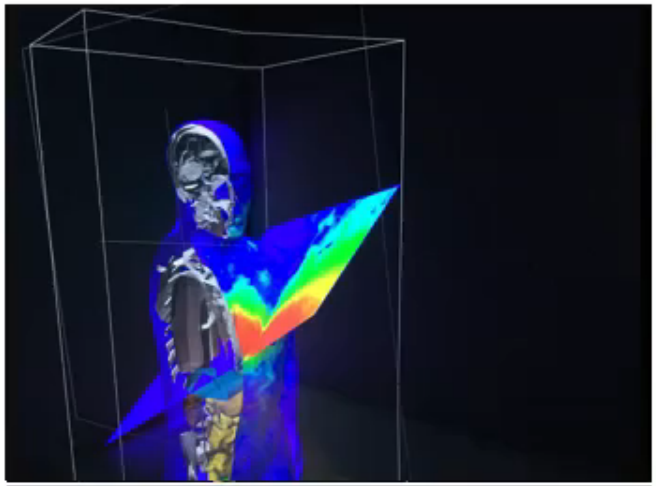
embrace limitations

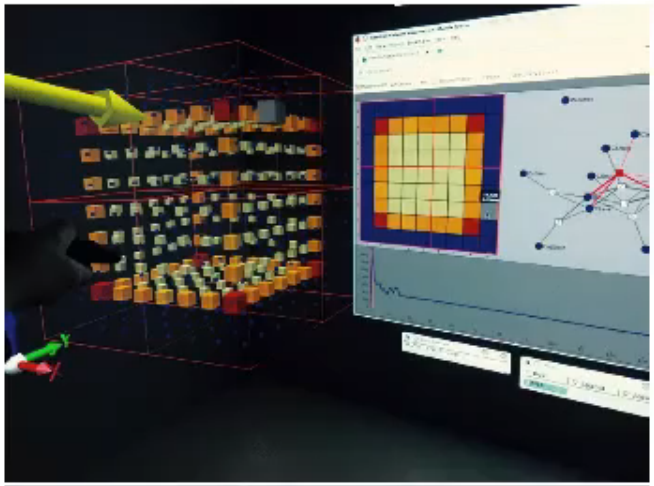
as they drive creativity

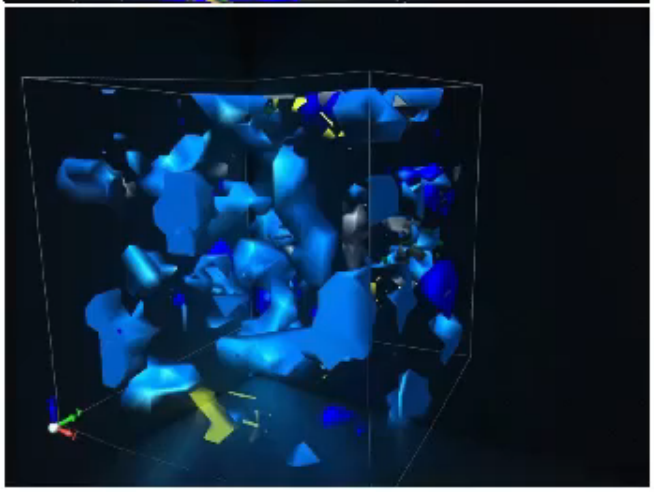

“If you are working on something exciting that you really care about, you don't have to be pushed. The vision pulls you.”

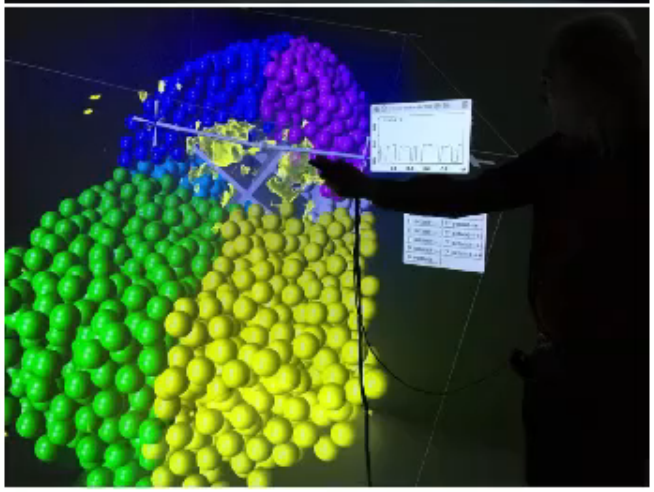
Immersive Virtual Measurement Techniques at NIST

Steven Satterfield, Judith Terrill, John Hagedorn, Wes Griffin, William George
National Institute of Standard and Technology







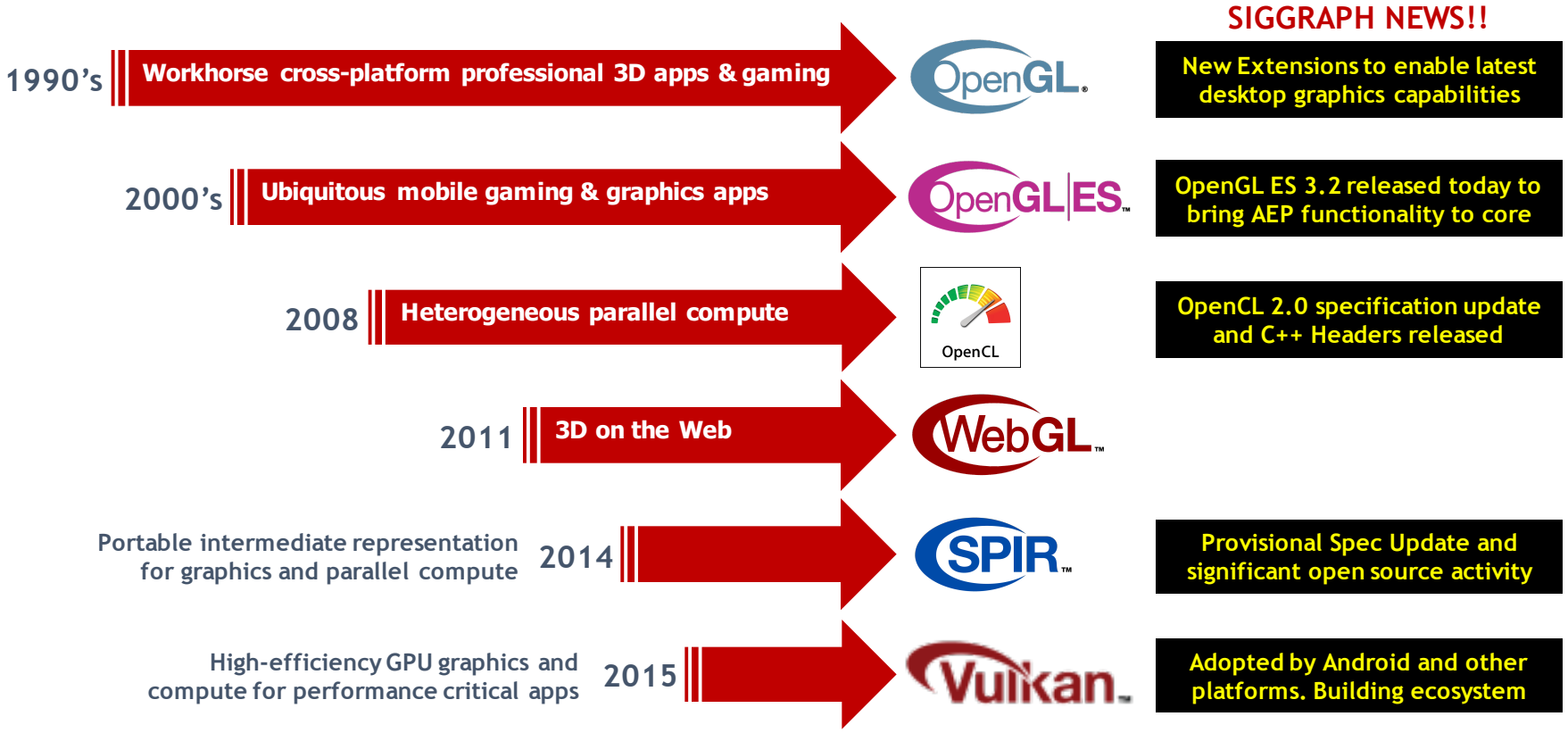


OpenGL history

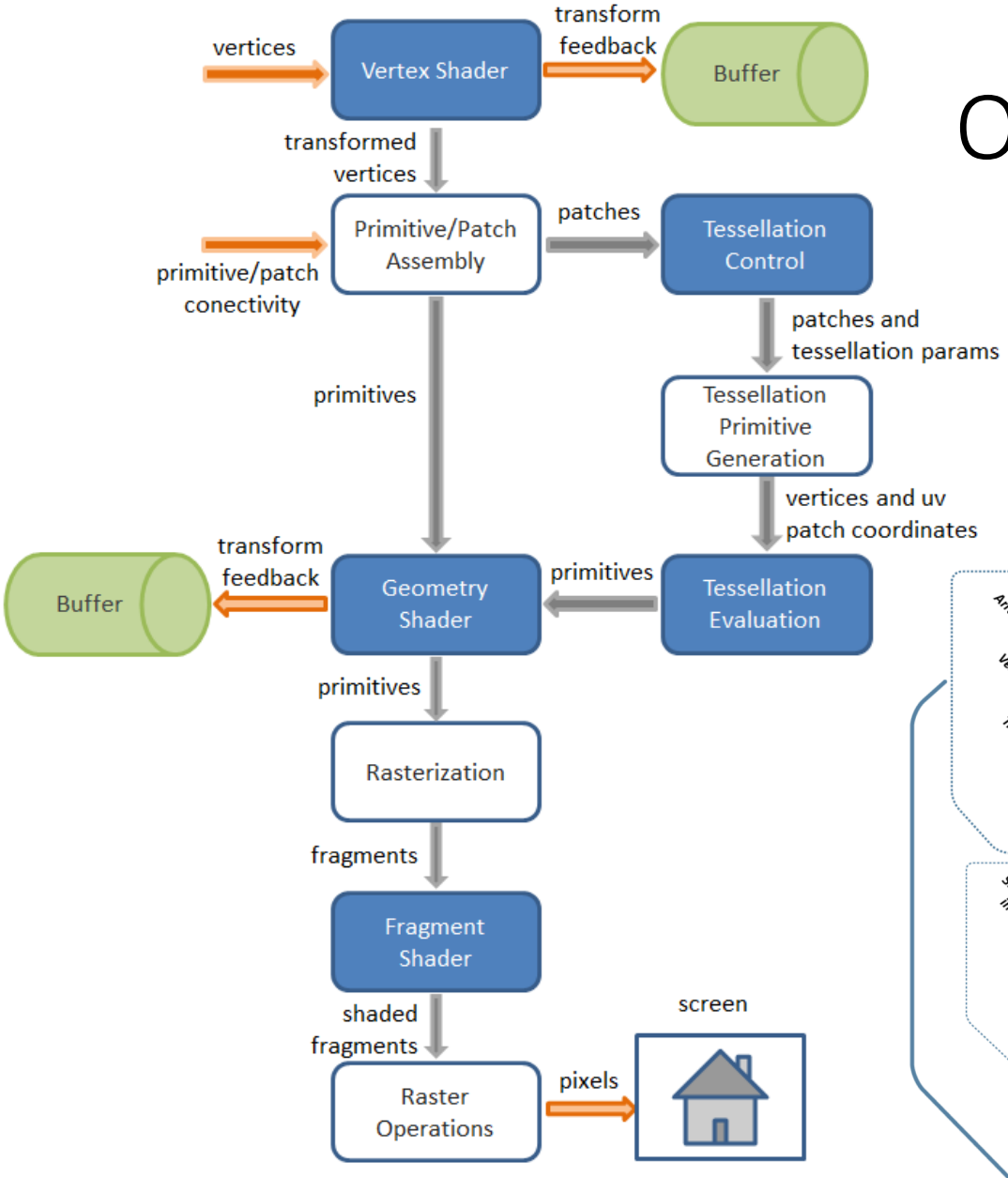
- **OpenGL 1.0**
  - Released in 1992 by OpenGL architectural review board (OpenGL ARB), by Mark Segal and Kurt Akeley
- **OpenGL 2.0**
  - Released on September 7, 2004. Support Shader language GLSL
- **OpenGL 3.0**
  - Released on July 11, 2008
- **OpenGL 4.0**
  - Released on March 11, 2010
- Current **OpenGL 4.5**
  - Released in August 2014

Open Standards

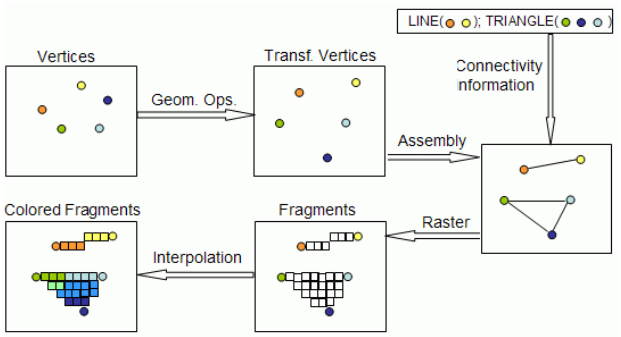
KHRONOS GROUP

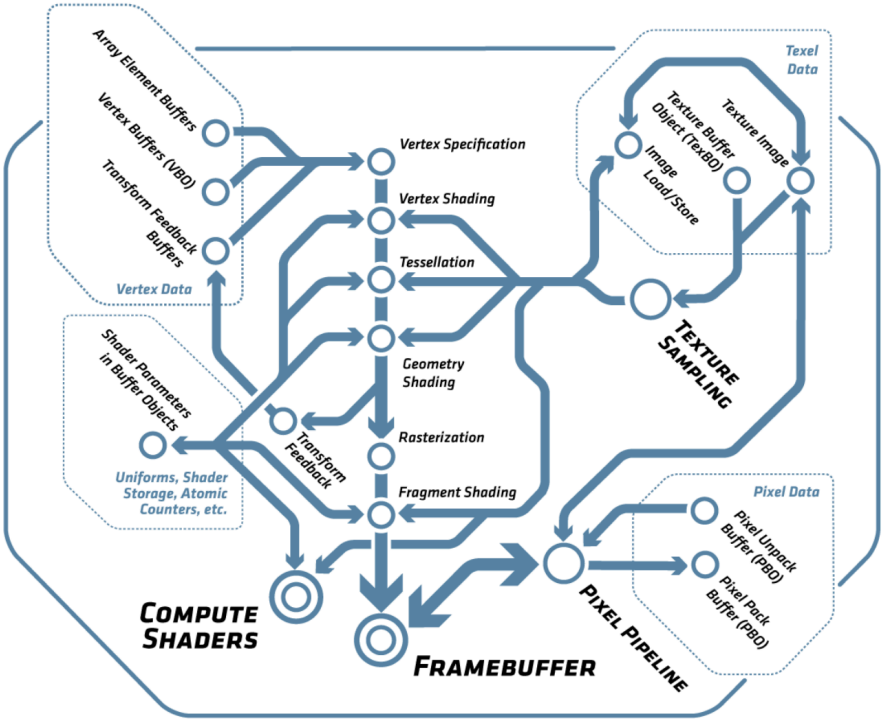


OpenGL pipeline

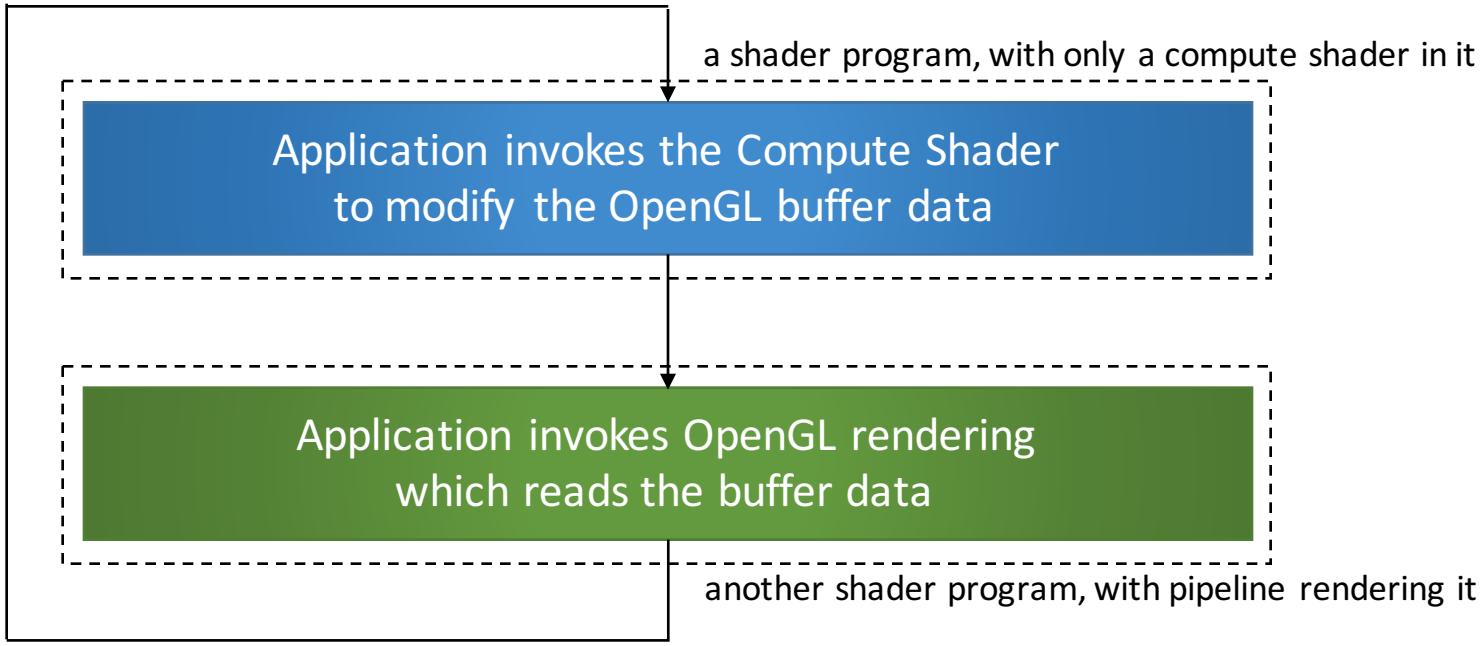


Source: http://hannahl.com/wp-content/uploads/2015/03/pipeline4.png





OpenGL Compute Shader



Compute Shaders vs OpenCL

- OpenCL requires installing separate driver and libraries
- Compute Shaders use familiar GLSL
- Compute Shaders use same context as the OpenGL rendering pipeline
- Using OpenCL is more complex: requires a lot of setup (queries, platform, devices, kernels, etc.)
- Use OpenCL for big stuff, OpenGL for computing that interacts with graphics

Graphics Pipeline

- **Scenes** are composed of models in two or three dimensional space
- **Models** are composed of primitives (such as vertex, triangle) supported by rendering system
- **Graphics pipeline** is the steps of conversion from **model** to **scene** and ultimately to an **image**

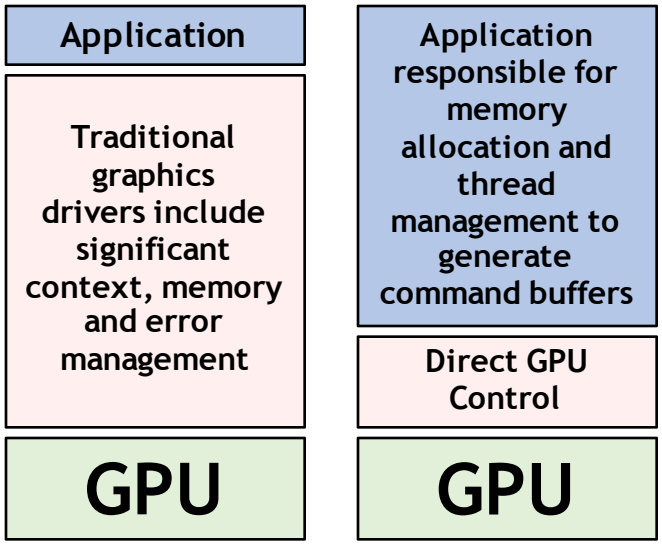
Vulkan Explicit GPU Control

Complex drivers lead to driver overhead and cross vendor unpredictability

Error management is always active

Driver processes full shading language source

Separate APIs for desktop and mobile markets



Simpler drivers for low-overhead efficiency, predictability and cross vendor consistency

Layered architecture so validation and debug layers can be unloaded when not needed

Run-time only has to ingest SPIR-V intermediate language

Unified API for mobile, desktop, console and embedded platforms

Vulkan delivers the maximized performance and cross platform portability needed by sophisticated engines, middleware and apps

OpenGL Shading Language (GLSL)

- Brings power and flexibility to programmers interested in creating modern, interactive, and graphical programs
- GLSL programs must be a part of larger OpenGL program (can't stand by themselves)
- Every OpenGL program will utilise several GLSL programs (**shader programs**)
- Each shader executes within a different section of the OpenGL pipeline
- Each shader runs on GPU and implement the algorithms for lighting and shading, but can be also used to perform animation, tessellation and general computation

Profiles – Core vs Compatibility

- OpenGL 3.0 introduced **depreciation model**
  - Functions or features can be marked depreciated for future removal
  - glBegin/glEnd was removed in OpenGL 3.1
- To maintain backwards compatibility use **compatibility profile**
- To use a particular version of OpenGL, use **core profile**
- There is also **forward compatible** profile (with all deprecated functions removed)

GLFW

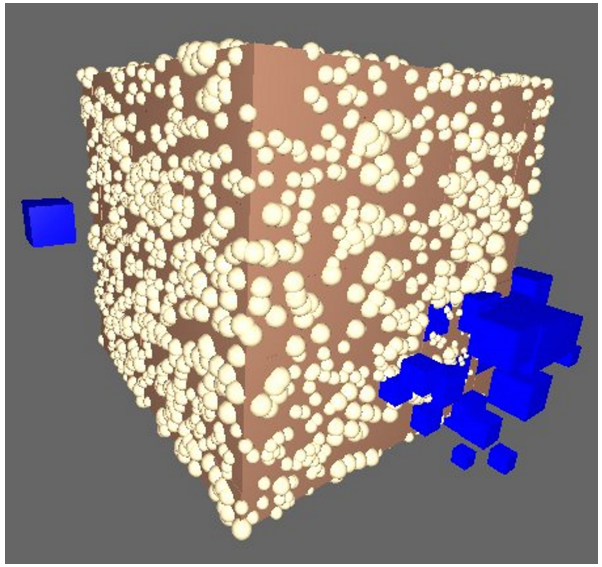
http://www.glfw.org/

- **GLFW** is an Open Source, multi-platform library for creating windows with OpenGL contexts and receiving input and events. It is easy to integrate into existing applications and does not lay claim to the main loop.
- **GLFW** is written in C and has native support for Windows, OS X and many Unix-like systems using the X Window System, such as Linux and FreeBSD.
- Enable forward compatible, 4.4 core profile:

```
glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 4 );
glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 4 );
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
GLFWwindow *window = glfwCreateWindow(640, 480, "Title", NULL, NULL);
```

GLFW

http://www.glfw.org/



```c
#include <GLFW/glfw3.h>

int main()
{
    GLFWwindow* w;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    w = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!w)
    {
        glfwTerminate();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(w);

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(w))
    {
        /* Render here */

        /* Swap front and back buffers */
        glfwSwapBuffers(w);

        /* Poll for and process events */
        glfwPollEvents();
    }

    glfwTerminate();

    return 0;
}
```

Function pointers loader

- The **OpenGL ABI** (application binary interface) is frozen to OpenGL 1.1 on Windows.
- Need to get access to functions pointers at runtime.
- GLEW (OpenGL Extension Wrangler) provides functions pointers, however:
  - It needs to be compiled separately
  - Does not support core profiles properly
  - Includes large header file from all OpenGL versions

Use function pointers loader

- Use OpenGL Loader Generator https://bitbucket.org/alfonse/glloadgen/wiki/Home
  - Use Lua http://luabinaries.sourceforge.net/
  - Execute C or C++ style
    - lua LoadGen.lua -style=pointer_c -spec=gl -version=4.4 -profile=core core_4_4
    - lua LoadGen.lua -style=pointer_cpp -spec=gl -version=4.4 -profile=core core_4_4
  - Two files are created:
    - gl_core_4_4.c(pp) and gl_core_4_4.h(pp)
- After OpenGL context is created, need to call:

```
if (ogl_LoadFunctions() == ogl_LOAD_FAILED) {
   // kill window, etc.
}
// call other functions
```

GLEW

http://glew.sourceforge.net/

- GLEW = The OpenGL Extension Wrangler Library
- One-stop-shop for API support for all OpenGL extension APIs
- Provides API support for all new extensions, including OpenGL 4.5
- It needs initialization, after OpenGL context is created:

```
// start GLEW extension handler
glewExperimental = GL_TRUE;
if (glewInit() != GLEW_OK) {
    glfwTerminate();
    return -1;
}
```

OpenGL Mathematics

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

- OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the GLSL specifications
- Provides classes and functions designed with the same naming conventions and functionalities as GLSL, to be used also within your C++
- Platform independent

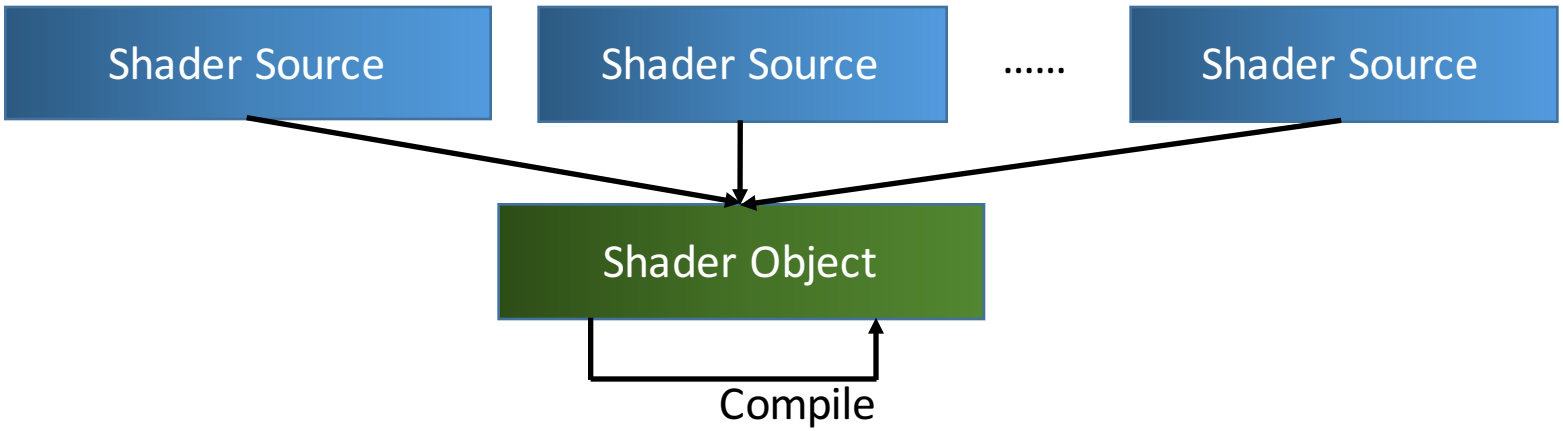
```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale, glm::perspective

glm::mat4 camera(float Translate, glm::vec2 const & Rotate)
{
    glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.f);
    glm::mat4 View = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -Translate));
    View = glm::rotate(View, Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
    View = glm::rotate(View, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
    return Projection * View * Model;
}
```

Get OpenGL version

```
// get OpenGL version information
//

const GLubyte* renderer = glGetString(GL_RENDERER); // get renderer string
const GLubyte* vendor = glGetString(GL_VENDOR); // get vendor string
const GLubyte* version = glGetString(GL_VERSION); // version as a string
const GLubyte* glslVersion = glGetString(GL_SHADING_LANGUAGE_VERSION); // GLSL version as a string

GLint major, minor;
glGetIntegerv(GL_MAJOR_VERSION, &major);
glGetIntegerv(GL_MINOR_VERSION, &minor);

std::cout << "GL Vendor: " << vendor << std::endl;
std::cout << "GL Renderer: " << renderer << std::endl;
std::cout << "GL Version: " << version << std::endl;
std::cout << "GL Version: " << major << "." << minor << std::endl;
std::cout << "GLSL Version: " << glslVersion << std::endl;

// get the extensions
//

GLint nExtensions;
glGetIntegerv(GL_NUM_EXTENSIONS, &nExtensions);
for (int i=0; i<nExtensions; i++)
{
   std::cout << glGetStringi(GL_EXTENSIONS, i) << std::endl;;
}
```

Compiling a shader

- The GLSL compiler is built into OpenGL library
- Shaders can be only compiled within the context of running OpenGL program
- However, OpenGL 4.1 allows to save compiled shaders to a file (to avoid overheads)
- Compiling a shader involves creating a shader object, and asking the shader object to compile the code



Compiling a shader

```
enum GLSLShaderType
{
    VERTEX = GL_VERTEX_SHADER,
    FRAGMENT = GL_FRAGMENT_SHADER,
    GEOMETRY = GL_GEOMETRY_SHADER,
    TESS_CONTROL = GL_TESS_CONTROL_SHADER,
    TESS_EVALUATION = GL_TESS_EVALUATION_SHADER,
    COMPUTE = GL_COMPUTE_SHADER
};
```

- Example basic.vert vertex “pass through” shader:

```
#version 440

in vec3 VertexPosition;
in vec3 VertexColour;

out vec3 Colour;

void main()
{
    Colour = VertexColour;
    gl_Position = vec4(VertexPosition, 1.0);
}
```

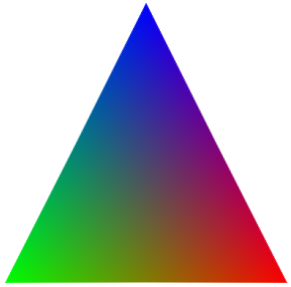
  - input attributes **VertexPosition** and **VertexColour** are passed to the fragment shader via the output variables **gl_Position** and **Colour**

- Build shell for OpenGL program using GLFW, GLUT, FLTK, QT, etc.

- Compile the shader:

```
// create shader object
GLuint shaderHandle = glCreateShader(type);

// copy the source into the shader object
const char *c_code = source.c_str();
glShaderSource(shaderHandle, 1, &c_code, NULL);

// compile the shader
glCompileShader(shaderHandle);

// verify the results
int result;
glGetShaderiv(shaderHandle, GL_COMPILE_STATUS, &result);

if (result == GL_FALSE) {
    int len, slen;
    glGetShaderiv(shaderHandle, GL_INFO_LOG_LENGTH, &len);

    GLchar* log = new GLchar[length];
    glGetShaderInfoLog(shaderHandle, length, &slen, log);
    std::cout << "log:\n" << log << std::endl;
    delete[] log;

    // add more error handling
}
else {
    glAttachShader(m_uiHandle, shaderHandle);
}
```

Linking a shader program



- After compiling shaders, we need to link them to a shader program.

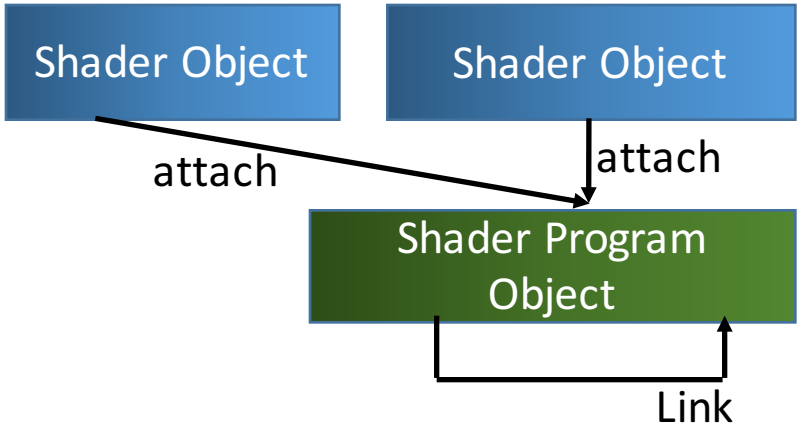
```
GLuint m_uiHandle;

m_uiHandle = glCreateProgram();
if (m_uiHandle == 0)
{
    throw GLSLProgramException("error ...");
}
```

- Linking involves making the connections between input/output variables and appropriate locations in the OpenGL enviornment.
- We use handles to vertShader and fragShaders.

```
#version 440
out vec4 FragmentColuor; //fragment shader output
void main()
{
    FragmentColuor = vec4(1, 0, 1, 1);
}
```

- Link the shader program.

```
// attach shaders to the program object
glAttachShader(m_uiHandle, vertexShaderHandle);
glAttachShader(m_uiHandle, fragmentShaderHandle);

// link the program
glLinkProgram(m_uiHandle);

// ...

// install the program into OpenCL pipeline
glUseProgram(m_uiHandle);
```



Sending data to a shader

using vertex attributes and vertex buffer objects

- Vertex shader

```
#version 440

layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColour;

out vec3 Colour;

void main()
{
   Colour = VertexColour;
   gl_Position = vec4(VertexPosition, 1.0);
}
```

- Fragment shader

```
#version 440

in vec3 Colour;

out vec4 FragmentColuor;

void main()
{
   FragmentColuor = vec4(Colour, 1.0);
}
```

- Vertex information must be provided using generic vertex attributes, usually with conjunction with (vertex) **buffer objects**
- Per-vertex input attributes are defined using GLSL qualifier **in**
- Main program needs to supply the data for two attributes for each vertex (map the polygon data to these variables)
- Output variable **Colour** is sent to the fragment shader
- The attribute **VertexPosition** is passed to the built-in output variable **gl_Position**
- Fragment shader gets **Colour** and outputs it to **FragmentColour** (output variable)

Sending data to a shader

using vertex attributes and vertex buffer objects

```
GLSLProgram program_demo;

float positionData_demo[] = {
   -0.8f, -0.8f, 0.0f,
    0.8f, -0.8f, 0.0f,
    0.0f,  0.8f, 0.0f
};

float colourData_demo[] = {
   1.0f, 0.0f, 0.0f,
   0.0f, 1.0f, 0.0f,
   0.0f, 0.0f, 1.0f
};

// vertex array object
GLuint vao_demo;

// buffer objects
GLuint vbo_demo[2];
```

```
program_demo.compileShader(vs_demo, GLSLProgram::VERTEX);
program_demo.compileShader(fs_demo, GLSLProgram::FRAGMENT);

program_demo.bindAttribLocation(0, "VertexPosition");
program_demo.bindAttribLocation(1, "VertexColour");

program_demo.link();

glGenBuffers(2, vbo_demo);
GLuint posBufferHandle = vbo_demo[0];
GLuint colBufferHandle = vbo_demo[1];

// populate buffers
// position
glBindBuffer(GL_ARRAY_BUFFER, posBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), positionData_demo, GL_STATIC_DRAW);
// colour
glBindBuffer(GL_ARRAY_BUFFER, colBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), colourData_demo, GL_STATIC_DRAW);

// create and bind vertex array objects = store relationship
// between buffers and input attributes
glGenVertexArrays(1, &vao_demo);
glBindVertexArray(vao_demo);

// enable vertex attribute arrays
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

// map index 0 to position buffer
glBindBuffer(GL_ARRAY_BUFFER, posBufferHandle);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte*)NULL);

glBindBuffer(GL_ARRAY_BUFFER, colBufferHandle);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte*)NULL);

glBindVertexArray(0);
```

```
program_demo.use();
glBindVertexArray(vao_demo);
glDrawArrays(GL_TRIANGLES, 0, 3);
program_demo.unUse();
```

Sending data to a shader

using uniform variables

- Vertex attributes were used to provide input to shaders
- Uniform variables is another way
- They are intended to be used for data that may change infrequently
- Well suited for matrices used for modeling, viewing and projective transformations
- Within a shader, they are read-only, but can be initialized by assigning constant value at the start
- They are held in uniform namespace for the entire shader program

```
#version 440

layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColour;

uniform mat4 RotationMatrix;

out vec3 Colour;

void main()
{
   Colour = VertexColour;
   gl_Position = RotationMatrix * vec4(VertexPosition, 1.0);
}
```

```
#include <glm\glm.hpp>
#include <glm\gtc\matrix_transform.hpp>

...

glm::mat4 rotationMatrix = glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0.0f, 0.0f, 1.0f));
GLuint location = glGetUniformLocation(program_demo.getHandle(), "RotationMatrix");
if (location >= 0)
   glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
angle += 0.025f;
```

Uniform blocks and uniform buffer objects

- Uniform block is a group of uniform variable defined within a structure

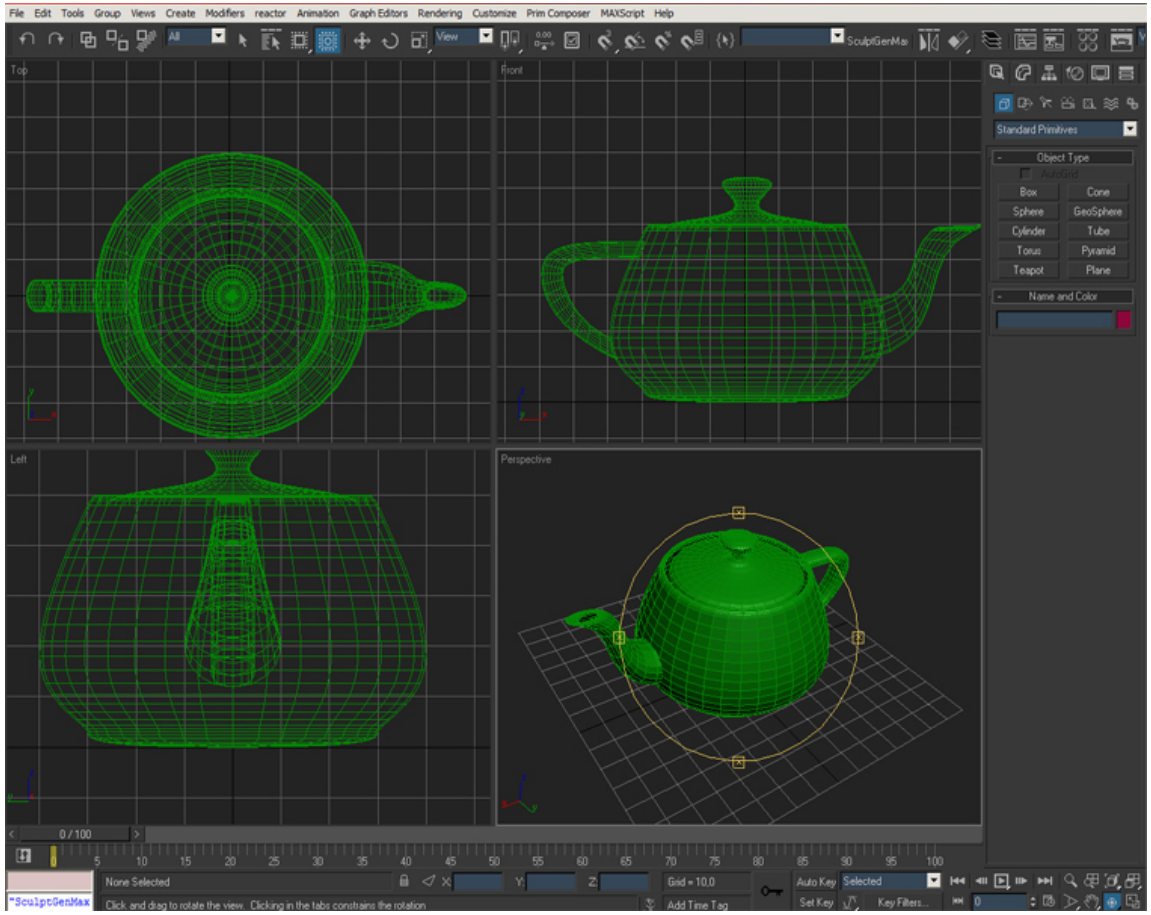
```
uniform BlobSettings {
    vec4 InnerColour;
    vec4 OuterColour;
    float RadiusInner;
    float RadiusOuter;
};
```

- The buffer object used to store the data for the uniforms is referred to as a uniform buffer object

```
#version 430

in vec3 TexCoord;
layout (location = 0) out vec4 FragColor;

layout (binding = 1) uniform BlobSettings {
  vec4 InnerColor;
  vec4 OuterColor;
  float RadiusInner;
  float RadiusOuter;
} Blob;

void main() {
    float dx = TexCoord.x - 0.5;
    float dy = TexCoord.y - 0.5;
    float dist = sqrt(dx * dx + dy * dy);
    FragColor =
       mix( Blob.InnerColor, Blob.OuterColor,
            smoothstep( Blob.RadiusInner, Blob.RadiusOuter, dist )
          );
}
```

```
#version 430

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexTexCoord;

out vec3 TexCoord;

void main()
{
    TexCoord = VertexTexCoord;
    gl_Position = vec4(VertexPosition,1.0);
}
```

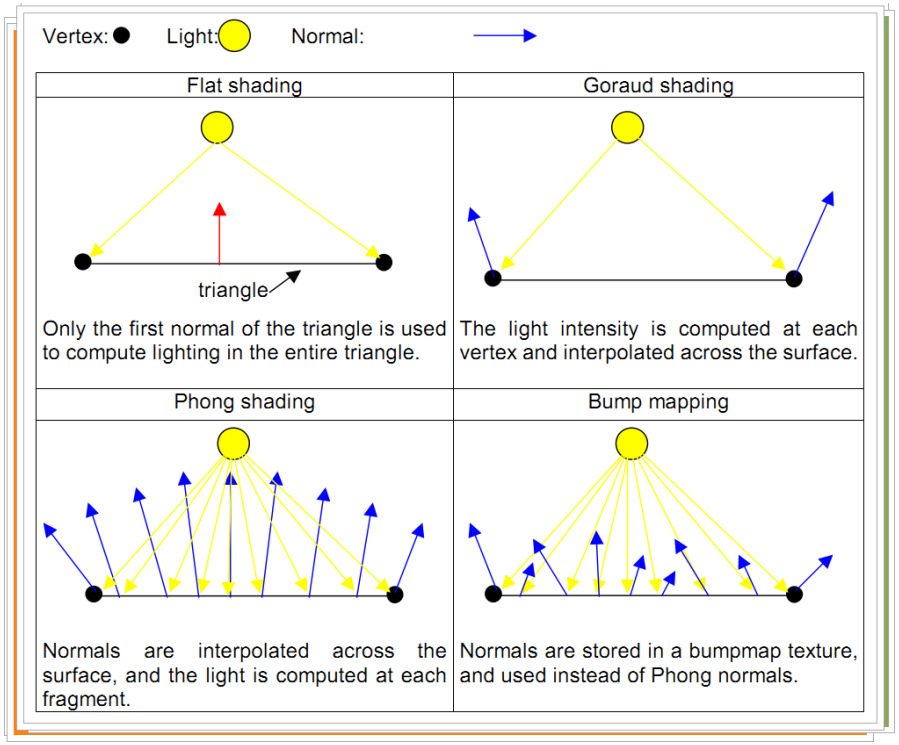
Load object from 3DS file

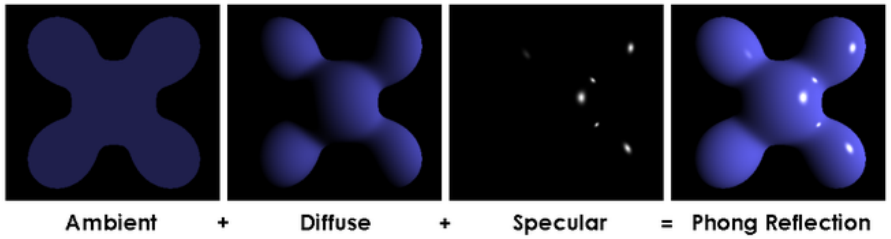



```
#define chunk_main                  0x4d4d
#define   chunk_mesh                0x3d3d
#define     chunk_material          0xafff
#define       chunk_matname         0xa000
#define       chunk_matambient      0xa010
#define       chunk_matdiffuse      0xa020
#define       chunk_matspecular     0xa030
#define       chunk_mattexture      0xa200
#define       chunk_mattexture_wm   0xa33e
                  ...
#define       chunk_matwiresize     0xa087
#define     chunk_object            0x4000
#define       chunk_trimesh         0x4100
#define         chunk_vertexlist    0x4110
#define         chunk_maplist       0x4140
#define         chunk_textureinfo   0x4170
#define         chunk_vertflags     0x4111
#define         chunk_facelist      0x4120
#define         chunk_matrix        0x4160
#define         chunk_meshcolor     0x4165
#define         chunk_smoothlist    0x4150
#define         chunk_facematerial  0x4130
#define       chunk_light           0x4600
#define         chunk_spotlight     0x4610
#define       chunk_camera          0x4700
#define         chunk_camranges     0x4720
#define   chunk_keyframer           0xb000
#define     chunk_kfhdr             0xb00a
#define     chunk_frames            0xb008
#define     chunk_nodeobject        0xb002
#define       chunk_nodehdr         0xb010
#define       chunk_nodeid          0xb030
#define       chunk_pivot           0xb013
#define       chunk_postrack        0xb020
#define       chunk_objhide         0xb029
#define       chunk_objrot          0xb021
#define       chunk_objscale        0xb022
#define     chunk_nodelight         0xb005
#define     chunk_nodecamera        0xb003
#define       chunk_fovtrack        0xb023
#define       chunk_rolltrack       0xb024
#define     chunk_nodecameratarget  0xb004
```

Phong shading

- Sum of three components: ambient, diffuse and specular
- Ambient: represents light that illuminates all surfaces equally and reflects equally in all directions
- Diffuse: models a rough surface that scatters light in all directions
- Specular: use to model the shininess of a surface

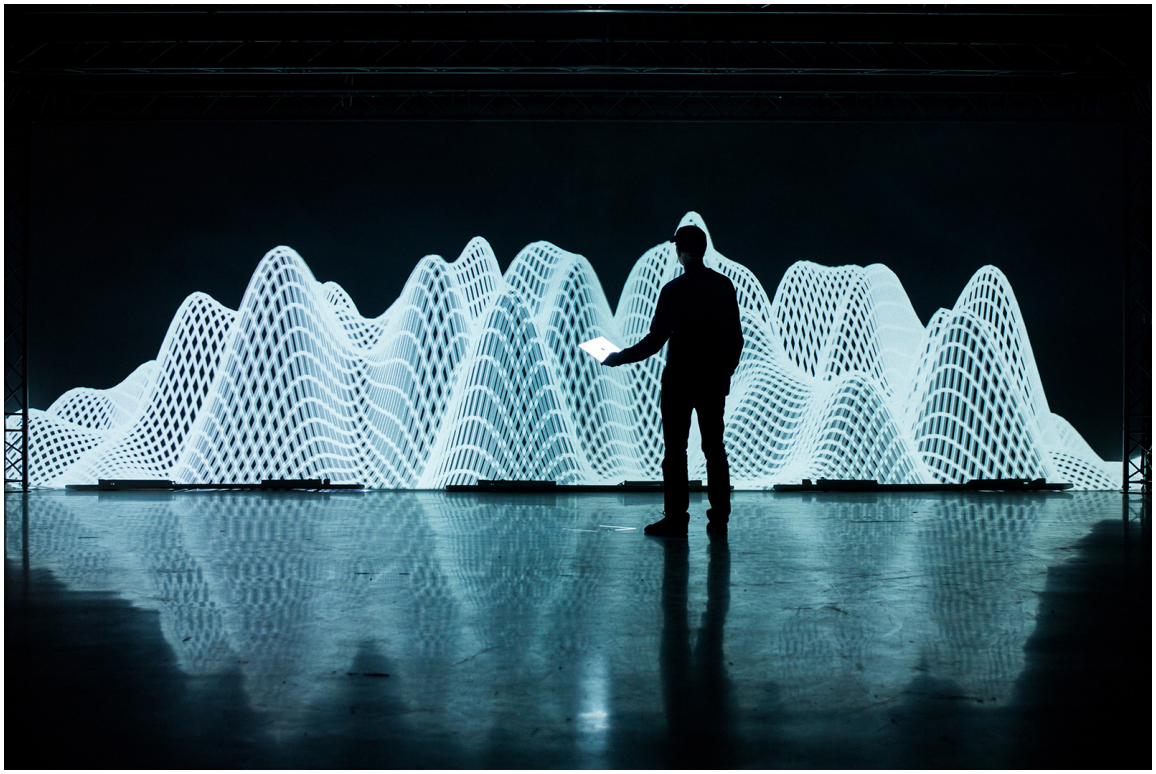




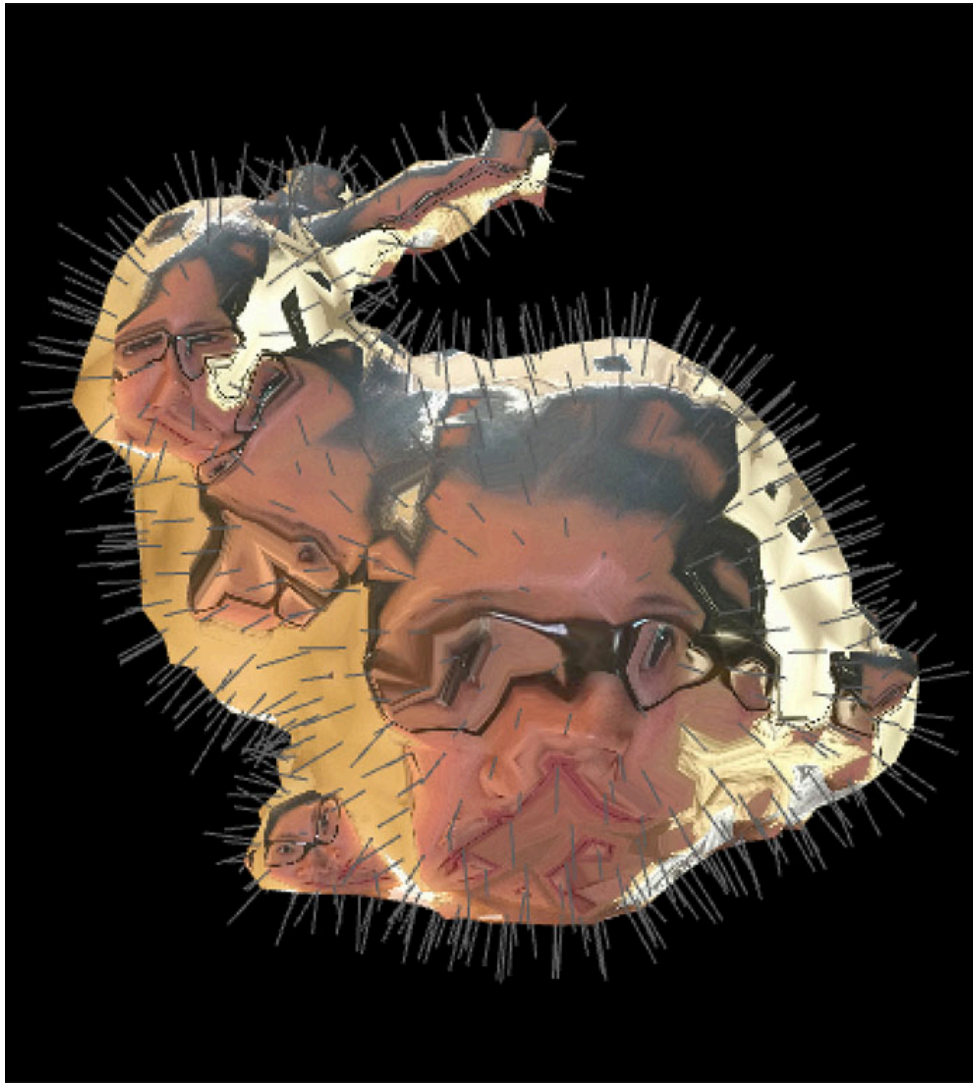
openFrameworks

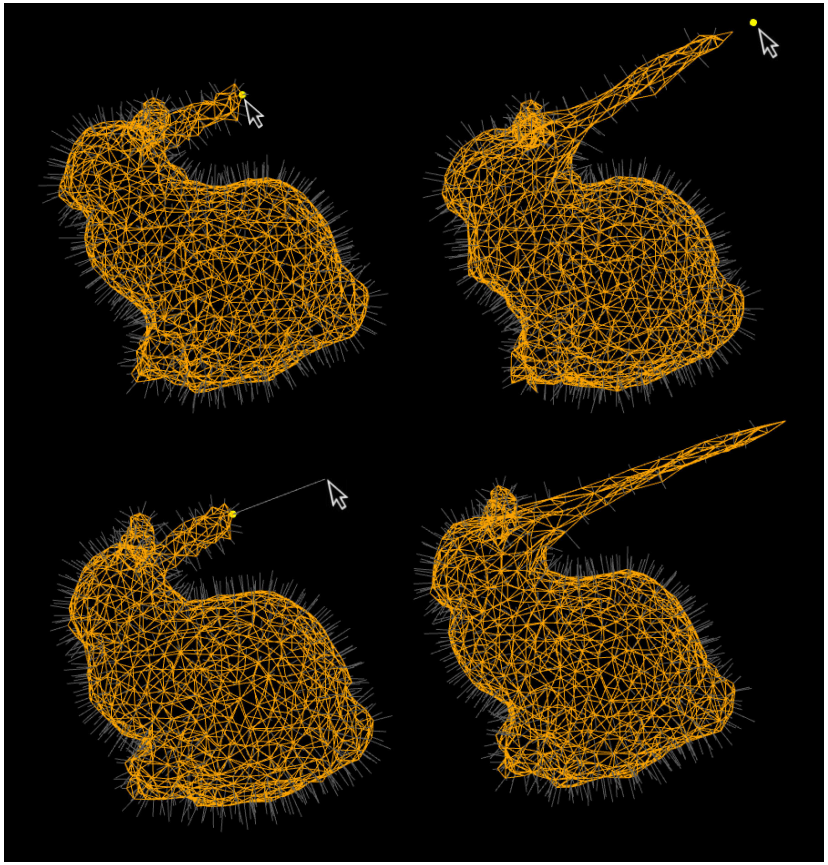
http://openframeworks.cc/

- openFrameworks is an open source C++ toolkit for creative coding

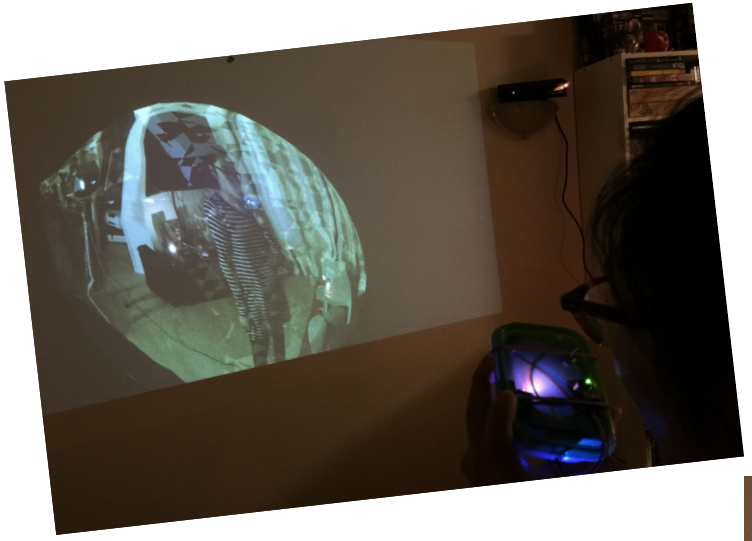


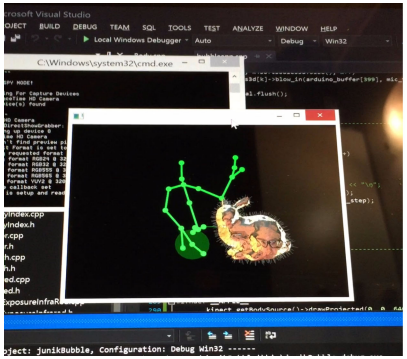
Interactive bunny

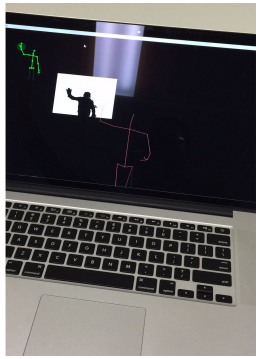


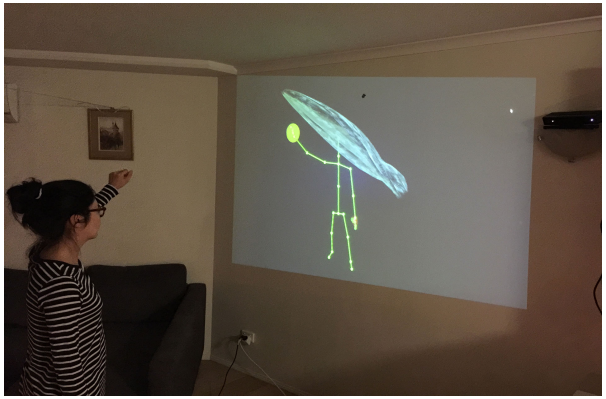


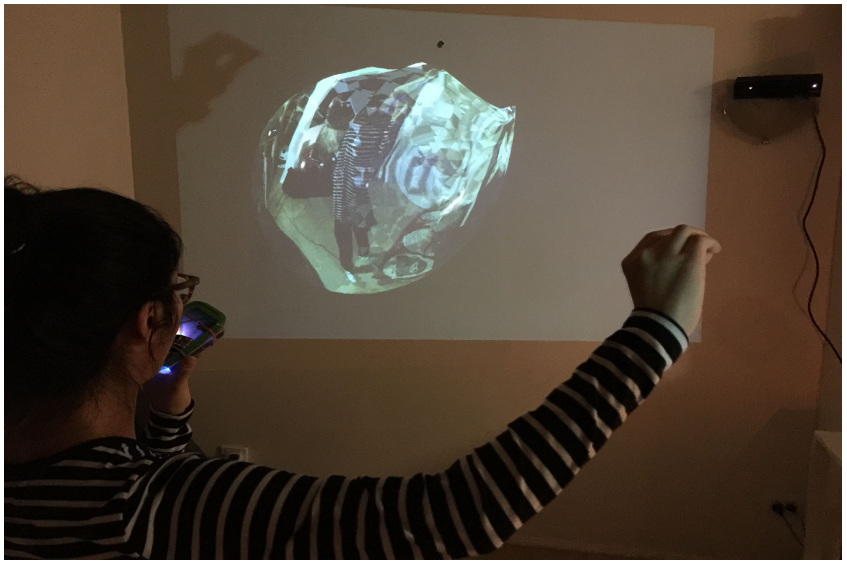
Reflecting / interacting / mirroring



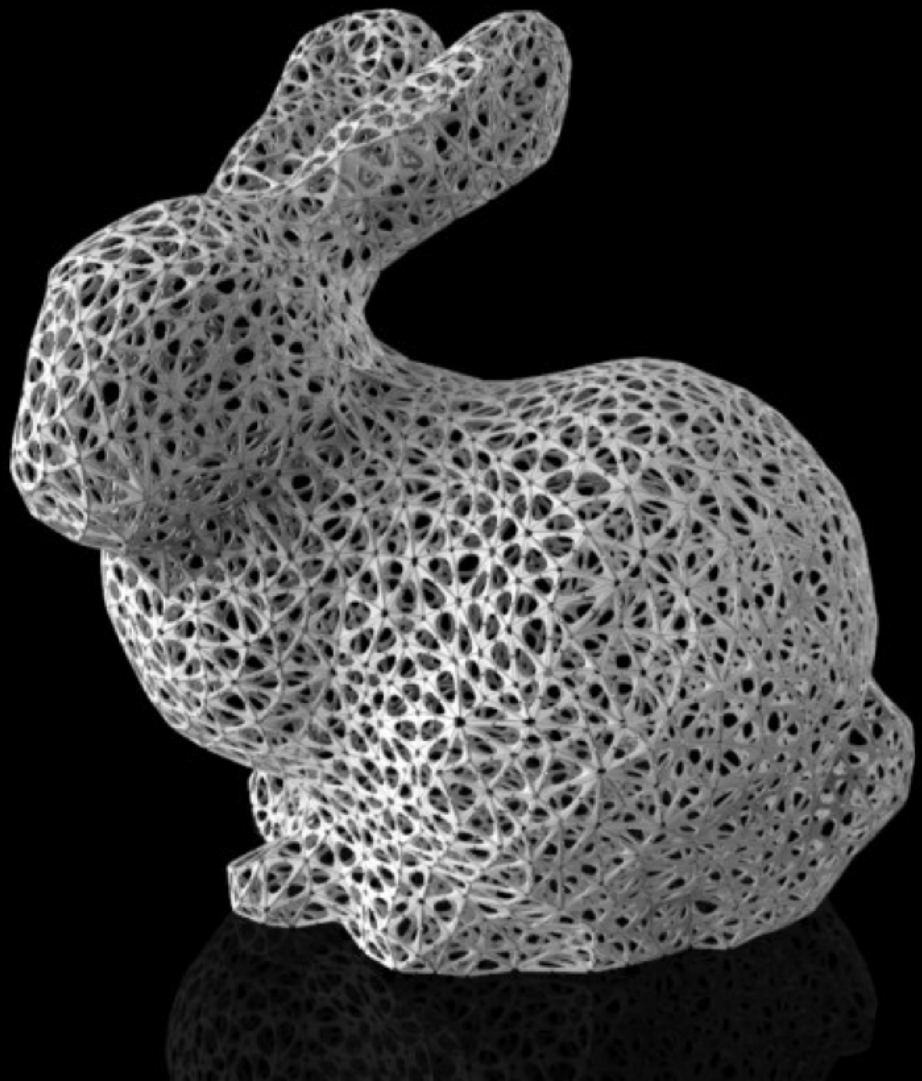








Where Science meets Art…



Let’s move to geometry shaders and tessellation shaders