Vertex and fragment programs

Jon Hjelmervik

email: jonmi@ifi.uio.no

Fixed function transform and lighting

- Each vertex is treated separately
- Affine transformation transforms the vertex by matrix multiplication
- Lighting
  - Determines the color of each vertex.
  - Calculated using normal vector and light direction/position.
  - Can be manipulated by light parameters, light model, material properties and position
- Texture parameter(s) are transformed using texture matrices.

Advanced vertex transformations

- Real time applications store vertex data on graphics memory, therefore all vertex transformations must be done at the graphics processor (GPU).
- Animations of a human body need to modify vertices in a non linear way depending on bones or control points.
- Morphing is used for animation, and uses a convex combination of two or more objects to create intermediate objects.

- Extending the fixed function API to handle all such applications would lead to a too messy interface.

Morphing at the CPU, using C++

- Assuming we use a vector library, morphing could be written as:

```
vec4 morphPosition(vec4 vertexPos1, float weight1,
                   vec4 vertexPos2, float weight2)
{
  vec4 Pos=weight1*vertexPos1;
  Pos+=weight2*vertexPos2;
  return Pos;
}
```

Morphing and lighting using openGL shading language

```
attribute vec4  vertexPos1;
attribute vec4  vertexPos2;
uniform float weight1;
uniform float weight2;
void main(void)
{
  vec4 Pos=weight1*vertexPos1;
  Pos+=weight2*vertexPos2;  // Morph position
  vec3 Norm=weight1*vertexNorm1;
  Norm+=weight2*vertexNorm2; // Morph normals
  Norm=normalize(Norm); //normalize the morphed normal
  vec4 ecPosition = gl_ModelViewMatrix * Pos; // Transform position to eyespace
  vec3 tnorm = gl_NormalMatrix * Norm;       // Transform normal
  vec3 lightVec = normalize(gl_LightSource[0].position.xyz - vec3(ecPosition));
 // calculate vector from light to vertex in eye space
  gl_FrontColor.rgb=dot(tnorm,lightVec); // calculate color
  gl_Position = gl_ModelViewProjectionMatrix * Pos;
 // Transform position to window space
}
```

Vertex shaders in OpenGL shading language

- OpenGL shading language is a C-like programming for defining vertex shaders and fragment shaders.
- Vertex shaders takes two types of input
  - **Uniform** variables are variables that are constant for the entire triangle. Examples: modelview matrix and light position. Uniform variables cannot be set between glBegin and glEnd.
  - **Attribute** variables that differs from vertex to vertex. Examples: position, normal and texture coordinate.
- Vertex shaders **must** return vertex position transformed to window coordinates (ready to be projected).

Vertex shaders in OpenGL shading language (2)

- A vertex shader acts on one vertex at the time and is responsible for the T&L part of the rendering pipeline, this includes:
  - Transforming the vertex into window space
  - Transforming the normal, and normalization
  - Lighting and calculating the color of the vertex
  - Generating/transforming texture coordinates

Matrix and vector data types

- The GPU is a vector processor, which uses vectors of length 4.
- vectors:
  - vec2, vec3 and vec4 are two, three and four component vectors respectively.
- The name of the components are given by one letter
- Three naming conventions can be used {x,y,z,w} {r,g,b,a} {s,t,p,q} where x, r and s is the first component in a vector.
- Swissling
  - vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
  - vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
  - vec4 dup = pos.rrgg; // dup = (1.0, 1.0, 2.0, 2.0)
  - pos.yx = vec2(1.0, 0.0); // pos = (0.0, 1.0, 3.0, 4.0)
  - vec3 tmp = pos.xrs; // not valid
- mat2, mat3 and mat4 are 2x2, 3x3 and 4x4 matrices respectively.

Commonly used built-in uniform variables

- Built-in variables are set by standard openGL calls.
- uniform mat4 gl_ModelNiewMatrix;
- uniform mat4 gl_ProjectionMatrix;
- uniform mat4 gl_ModelViewProjectionMatrix;
- uniform mat4 gl_NormalMatrix;

- uniform glLightSourceParameters gl_LightSource[gl_MaxLights];
  // array of structs containing light parameters

Commonly used built-in attributes

- built in attributes are set by standard openGL calls, such as glVertex() and glNormal()
- attribute vec4 gl_Color; // The color of the vetex
- attribute vec4 gl_Normal; // Vertex normal
- attribute vec4 gl_Vertex; // Vertex position
- attribute gl_MultiTexCoord0; // texture coordinate

Vertex shaders can not

- Any operation that requires knowledge about neighbors
- Polygon clipping.
- Generate new vertices or primitives.
- Set global data.
- Remove geometry (culling).

- A GPU transforms (shades) several vertices in parallel, therefore any operation requiring that the vertices are transformed in a specific order is impossible.

Morphing and lighting using openGL shading language revisited

```
attribute vec4  vertexPos1;
attribute vec4  vertexPos2;
uniform float weight1;
uniform float weight2;
void main(void)
{
  vec4 Pos=weight1*vertexPos1;
  Pos+=weight2*vertexPos2;  // Morph position
  vec3 Norm=weight1*vertexNorm1;
  Norm+=weight2*vertexNorm2; // Morph normals
  Norm=normalize(Norm); //normalize the morphed normal
  vec4 ecPosition = gl_ModelViewMatrix * Pos; // Transform position to eyespace
  vec3 tnorm = gl_NormalMatrix * Norm;       // Transform normal
  vec3 lightVec = normalize(gl_LightSource[0].position.xyz - vec3(ecPosition));
 // calculate vector from light to vertex in eye space
  gl_FrontColor.rgb=dot(tnorm,lightVec); // calculate color
  gl_Position = gl_ModelViewProjectionMatrix * Pos;
 // Transform position to window space
}
```

Fragment shaders

Fixed function texturing

- Simple openGL applications does one texture lookup based on the texture coordinate, and either multiplies, adds or replaces the input color by this value.
- More complex methods for combining textures are available using fixed functions, but the API is complex and the functions are not flexible.

Per pixel lighting and advanced texturing

- High quality rendering of complex models we must either calculate the lighting per pixel, or use many triangles.
- Realistic car-paint rendering requires complex light models, and per pixel lighting and reflection calculations.
- Toon shading, makes the scene look like a part of a cartoon.
- Bump mapping uses a texture to augment the normal vector, and uses the resulting vector for lighting calculations.
- Realistic skin rendering requires several texture lookups per pixel and complex calculations to combine the results.

Phong shading/normal map example

```
uniform sampler2D normalMap;
uniform vec3 lightVect; // Directional light, light vector in object space

void main(void)
{
  vec3 normal=texture2D(normalMap, gl_MultiTexCoord0,xy);
  normal = normalize(normal);
  gl_FragColor = gl_Color*dot(lightVect, normal);
}
```

OpenGL fragment shader

- A fragment shader is a programmable replacement for the texturing in fixed function pipeline.
- Fragment shaders takes two types of input
    - **Uniform** variables are variables that are constant for the entire triangle. Examples: modelview matrix and light position. Uniform variables cannot be set between glBegin and glEnd.
    - **Varying** variables are linearly interpolated between the vertices. Examples: color and texture coordinate. Varying variables are output from the vertices of the triangle, and therefore not accessible from the application.

Texture lookups in shader

- Both fragment shaders and vertex shaders may use texture lookups.
- Texture lookups require information of which texture/ texture unit to use. This information is located in samplers.
    - sampler1D one-dimensional texture
    - sampler2D two-dimensional texture
    - sampler3D three-dimensional texture
    - samplerCube cube map is a special texture where a 3D vector is used for texture lookups

Returning information from a fragment shader

- A fragment shader can return the following elements
  - discard, when a shader calls discard the fragment will not update the frame buffer.
  - gl_FragColor is the output color of the fragment.
  - gl_FragDepth, the fragment shader may change the depth value of the fragment by writing to this variable.

Input to a fragment shader

- Special input variables
  - vec4 gl_FragCoord, holds the window coordinates of the fragment. May be used to implement scissor test in a fragment shader.
  - bool gl_FronFacing is true for front facing triangles, and false for back faceing triangles.
- Commonly used built in varying variables
  - vec4 gl_Color
  - vec4 glTexCoord[gl_MaxTextureCoords]

Phong shading/normal map revisited

```
uniform sampler2D normalMap;
uniform vec3 lightVect; // Directional light, light vector in object space

void main(void)
{
  vec3 normal=texture2D(normalMap, gl_MultiTexCoord0,xy);
  normal = normalize(normal);
  gl_FragColor = gl_Color*dot(lightVect, normal);
}
```

Built-in functions

- Trigonometric functions:
  - radians, degrees, sin, cos, tan, asin, acos, atan
- Exponential functions :
  - pow, exp2, log2, sqrt, inversesqrt
- Regular functions :
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep
- Geometrical functions :
  - length, distance, dot, cross, normalize, ftransform, faceforward, reflect
- Matrix functions , vector relation functions , texture lookup functions , fragment processing functions and noise functions .

Branching using GeForce 6 series

- There are three different types of branching, the compiler chooses the type.
- **Compile time branching:** The compiler resolves the branch.
- **Dependent write:** All possible branches are calculated, and the result of the false ones are discarded (only used for fragment shaders).
- **True branching:** In a fragment shader true branching is very expensive unless many neighboring fragments go through the same branch.

Numerical simulation at GPU

- Simulation problem must be converted to a “geometric problem”
- Pro
  - More FLOPS per Dollar then CPU
  - Simulation at graphics hardware allows visualization embedded in simulation
- Cons
  - Less flexible than CPU
  - Less memory than CPU
  - Less bandwidth between “system” and GPU

Explicit schemes

- We have started investigating evolutionary PDEs, which can be solved using explicit schemes.

- When using explicit schemes the unknown(s) at each grid point is updated from its neighbors at previous time steps.

- Relatively simple to convert to a “geometric problem”.

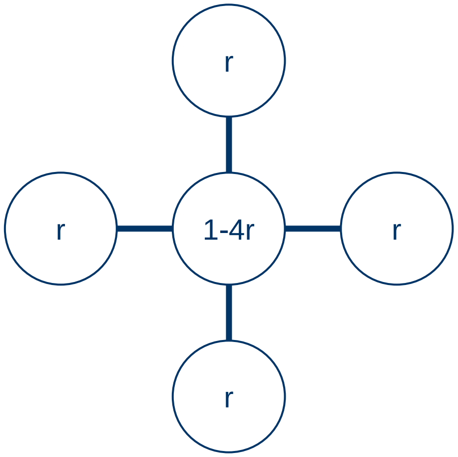
Heat Equation

- Very simple scheme.
- Implemented as a 1-pass algorithm.
- Scheme is the same as the Gauss filter used in image processing.
- The PDE is given as

$$u_t = u_{xx} + u_{yy}$$

and is discredited by a standard finite difference stencil to

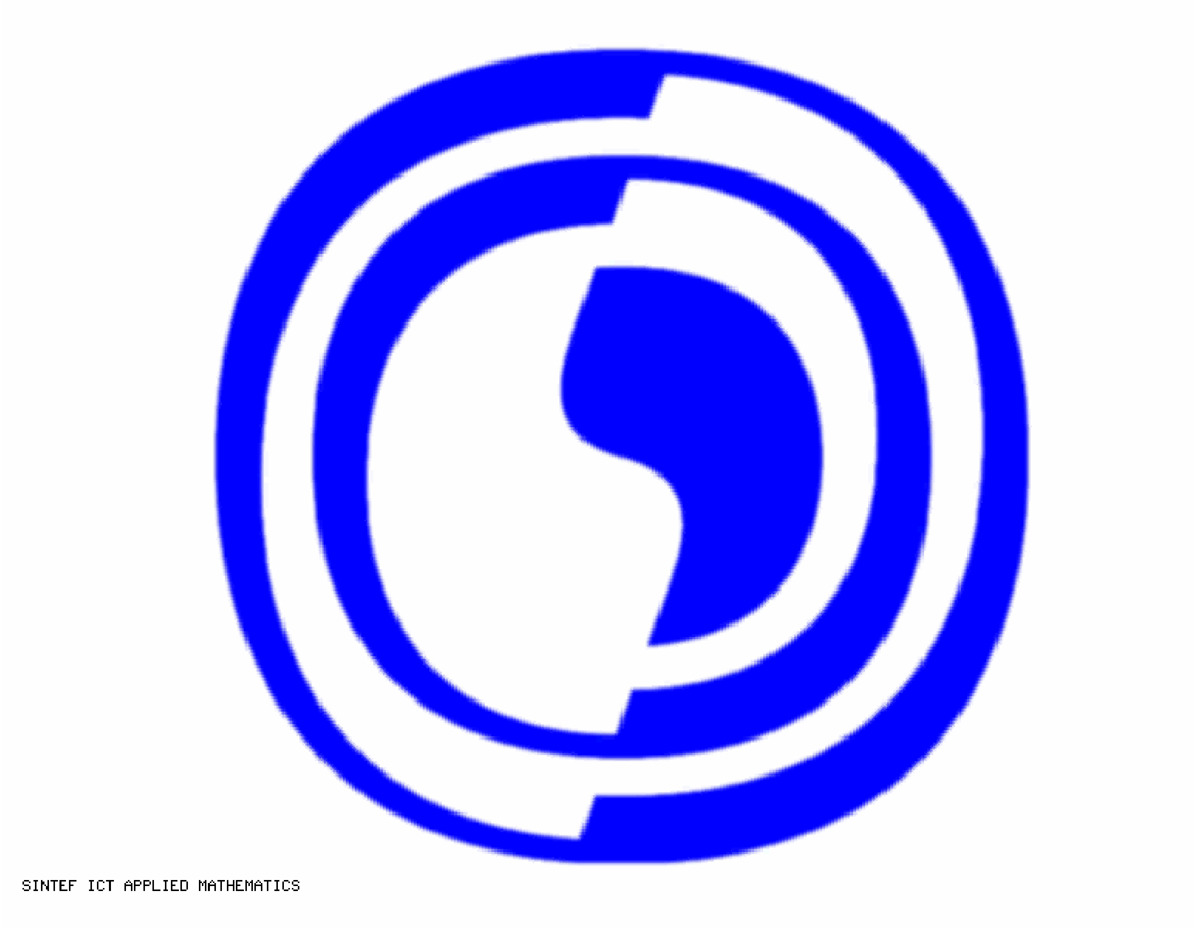
$$U_{i,j}^{n+1} = U_{i,j}^{n} + r\left(U_{i+1,j}^{n} + U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i,j+1}^{n} - 4U_{i,j}^{n}\right).$$



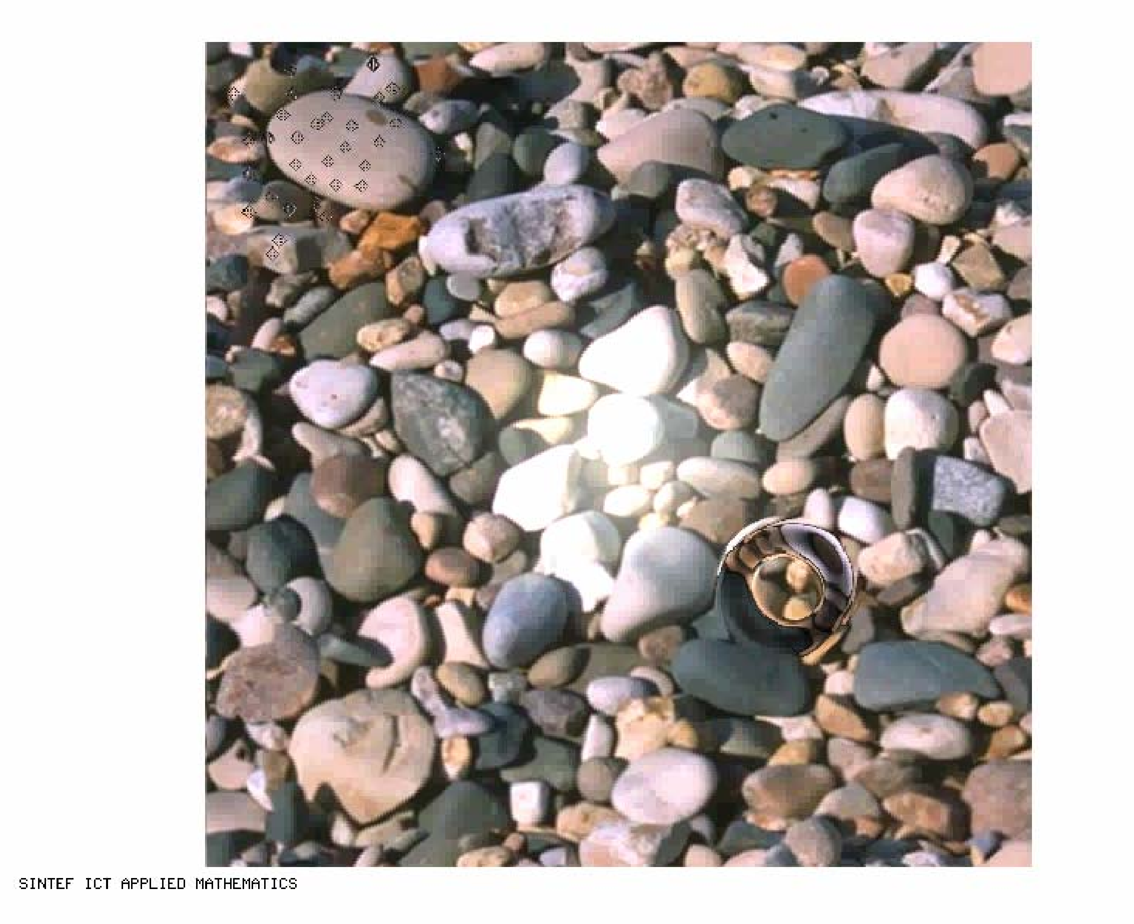
Implementation of heat equation

- Implemented as a fragment program.
- Uses two off screen pixel buffers, each with the same dimensions as the area of the simulation.
- Toggles drawing to one buffer, while reading the other as a texture.
- Render a quad covering the entire viewport.
- Each fragment reads the color of the pixels at the same position from the previous frame, and it’s neighbors.

Heat equation



Wave equation



Shallow water equation

SINTEF ICT APPLIED MATHEMATICS

References

- OpenGL Shading Language by Randy J. Rost
- www.opengl.org
- Shader Designer www.typhoonlabs.com

Bruk av shadere

1. Gi shader kildekode til OpenGL
2. Kompiler shader
3. Link sammen kompilerte shadere
4. Ta i bruk program

Merk: Kompilator ligger i driveren til grafikkortet.

Shader-objekter

Lage et shader-objekt:

```
shaderId = glCreateShaderObjectARB(shaderType);
```

shaderType:

`GL_VERTEX_SHADER_ARB`

`GL_FRAGMENT_SHADER_ARB`

Kildekode

Gi shader kildekode til OpenGL:

`glShaderSourceARB(shaderId, numStr, strings, length);`

- length kan settes til NULL hvis strengene er null-terminert

Kompilering

- Shader objekter blir kompilert ved:

  `glCompileShaderARB(shaderId);`

  Setter status parameter til GL_TRUE hvis suksess.

  Informasjon om kompilering kan fås tak i med:

  `glGetInfoLogARB(shaderId, bufferLen, strLen, buffer);`

Program objects

- Et program object er en kontainer for shader objects.
- Program objectet utgjør shaderene som må linkes sammen ved bruk.

```
programId = glCreateProgramObjectARB();
glAttachObjectARB(programId, shaderId);
glDetachObjectARB(programId, shaderId);
```

Sletting av objekter

- Shader objects og program objects slettes med:

  `glDeleteObjectARB(objectId);`

- Data for et shader object blir ikke slettet før objektet er frakoblet et program object.
- Data for et program object blir ikke slettet mens det er i bruk.

Linking

- Shaderene i et program object linkes med:

  `glLinkProgram(programId);`

- Informasjon om linkingen kan fås tak i med:

  `glGetInfoLogARB(programId, bufferLen, strLen, buffer);`

Bruke programmer

- For å ta i bruk et program kaller man:

```
glUseProgramObjectARB(programId);
```

- Hvis et program object er gyldig så blir det en del av gjeldende rendering modell.
- For å returnere til fixed function rendering modell så kaller man glUseProgramObjectARB(0);

Generic attributes

- Sette vertex attributes
  - glGetAttribLocationARB(programHandle_, name);
  - glVertexAttrib{1234}{fv}ARB(location, &attrib[0]);

- Sette vertex attribute pointers
  - glGetAttribLocationARB(programHandle_, name);
  - glBindAttribLocationARB(programHandle_, location, name);
  - glVertexAttribPointerARB(location, size, type, normalized, stride, pointer);

- Enable client state
  - glGetAttribLocationARB(programHandle_, name);
  - glEnableVertexAttribArrayARB(location);

Uniforms

- Sette uniforms
  - glGetUniformLocationARB(programHandle_, name);
  - glUniform{1234}{if}vARB(location, count, &constant[0]);
  - glUniformMatrix{234}fvARB(location, count, transpose, matrix)