Introduction to Computer Graphics

Juan Hernando Vieites
jhernando@fi.upm.es

ECAR 2012

1. Introduction to 3D Computer Graphics
   - Object modelling
   - Geometrical transformations
   - Lighting
   - Texturing and aliasing

2. Basic introduction to GLSL

1. Introduction to 3D Computer Graphics
    - Object modelling
    - Geometrical transformations
    - Lighting
    - Texturing and aliasing

2. Basic introduction to GLSL

3D Computer Graphics

Definition of Computer Graphics

- Computer Graphics is the area of Computer Science that has to do with the generation of synthetic images, also known as *rendering*.
- This presentation will be restricted to 3D graphics (but not necessarily photo-realistic images).

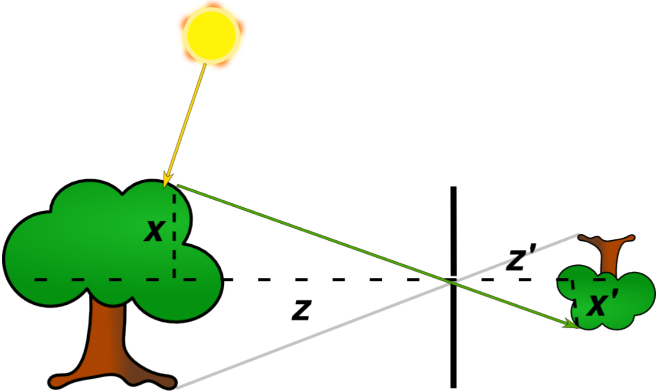
Computer Graphics & Visualization

- Research on 3D CG has its main driving force and application on the videogame and film industry.
- But graphics are also important to visualization as a basic image rendering tool (in 2D and 3D) and in more advanced rendering problems (e.g. volume rendering, automatic technical illustration).

References

- Computer Graphics: Principles and Practice, 2nd edition. Foley, Van Damm, Feiner, y Hughes. Addison-Wesley, 1.996
- Real time rendering (3rd edition). Tomas Akenine-Möller, Eric Haines, Naty Hoffman. A.K. Peters, 2008

The pinhole camera



$$x' = \frac{x z'}{z}$$

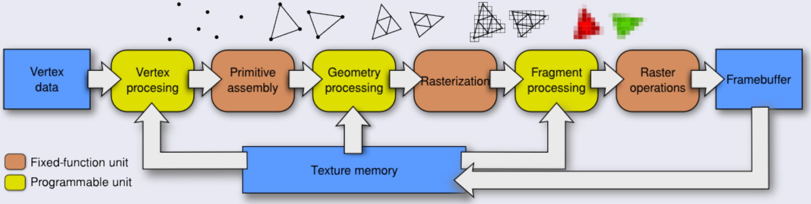
The graphics pipeline

Definition

Graphics pipeline is a common name to refer to the processing pipeline used in hardware accelerated rasterization to convert polygons (mainly triangles) into pixels.

Stages

The exact stages of a graphics pipeline depend on the author. A simplified view common to OpenGL and DirectX is shown below.



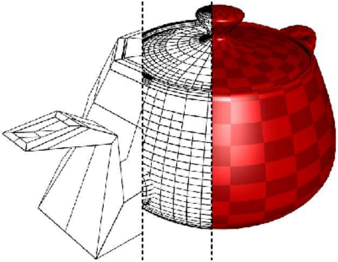
Object modelling

What is a model of an object

- A model is a description of an object with all the information needed for rendering.
- The image generation process is a simulation of the interactions between the light and the models.

Properties of object models

- The geometrical shape and location of elements.
- Topology.
- Features of the object elements (color, reflectance, transmittance, ...)



Geometrical models

Types of geometrical models

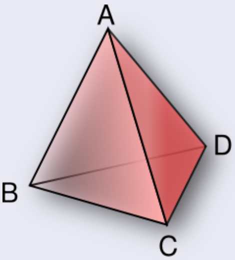
- Wireframe models.
- Parametric primitives: spheres, cylinders, ...
- Boundary representations (B-rep):
  - Polyhedral: polygonal and tetrahedral meshes.
  - Non-polyhedral: B-splines, NURBS, Bézier, subdivision surfaces.
- Scalar field representations: voxels, distance fields, . . .
- Others: Constructive solid geometry, spatial partition representations, . . .

In general, there is no one-size-fits all solution and even more, the most appropriate model and support data structures for simulations may not be the most appropriate for visualization.

Polygonal meshes

Representation

- Polygonal meshes can be represented in several ways.
- For GPU-based rendering the usual way is to use a vertex and index lists:



$$\begin{aligned}
\text{Vertices} &= [(0\ 0\ \tfrac{\sqrt{6}}{3}),\\
&\quad(-0.5\ \tfrac{\sqrt{3}}{4}\ 0),\\
&\quad(0\ -\tfrac{\sqrt{3}}{4}\ 0),\\
&\quad(0.5\ \tfrac{\sqrt{3}}{4}\ 0)]\\
\text{Faces} &= [A, B, C, A, C, D,\\
&\quad A, D, B, B, D, C]\\
\text{Strip} &= [B, A, C, D, B, A]
\end{aligned}$$

- Additional attributes can be provided at vertices or faces to be used for rendering.

Polygonal meshes

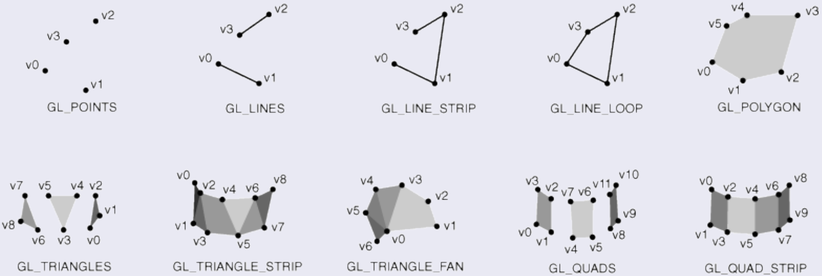
Observations

- Triangular meshes can be considered as the "machine code" of the models in computer graphics.
- In rasterization pipelines, most of the other representations cannot be directly rendered, So they are adaptively converted into meshes for display.
- When topology is needed, more complex data structures are required (e.g. DCEL, double connected edge list).

Geometrical primitives

Primitives

- The geometrical primitive are the basic geometrical objects that are directly supported by low-level graphics APIs.
- In OpenGL the primitives are: points, lines, line strips, triangles, triangle strips, triangle fans, quads, quad strips and polygons.



Vector spaces

- A vector space defines elements in $\mathbb{R}^n$ and operations between these elements and scalars: addition (vector-vector and scalar-scalar) and multiplication (scalar-scalar) and (scalar-vector).
- For a n dimensional space, a *base* of that space is given by n linearly independent vectors.

Affine spaces

- An affine space is a vector space extended with the notion of point.
- Subtraction of two points gives a vector and a point plus a vector gives a point.
- An affine space is defined by a vector base and an origin.

Linear transformations on $\mathbb{R}^3$

Definition and properties

- A linear transformation T of a vector in a vector space is a closed operation that satisfies:

$$T(\alpha \boldsymbol{v} + \beta \boldsymbol{u}) = \alpha T(\boldsymbol{v}) + \beta T(\boldsymbol{v})$$

- All linear transformations in $\mathbb{R}^3$ can be stated as a composition of: rotations, scalings, symmetries and shears.
- In $\mathbb{R}^3$, linear transformations can be represented as 3×3 matrices.
- Transforming a vector is multiplying it by a matrix: $\boldsymbol{v'} = \boldsymbol{M}\boldsymbol{v}$.
- Note that, in general, transformations are not commutative, as neither is the matrix product.

Affine transformations

Translations

- In the previous list of transformations there was one missing: translation.
- Translation is not a linear transformation!

Affine transformations

- An affine transformation is a linear transformation followed by a translation:

$$\boldsymbol{v}' = \boldsymbol{M}\boldsymbol{v} + \boldsymbol{p}$$

- Affine transformations only make sense for points. Vectors are directions so they should be invariant under translation.

Homogeneous coordinates

Translations in matricial form

- The addition of two vector $\boldsymbol{v}$ and $\boldsymbol{t}$ can be represented as matrix vector product if vectors are extended with an additional component with value 1.
- For a 2D translation the 3×3 matrix is:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} (v_x \ v_y \ 1) = (v_x + t_x \ v_t + t_t 1)$$

Homogeneous coordinates

- In homogeneous coordinates points are expressed in the form $(p_x \ p_y \ p_z \ 1)$ while vectors are $(v_x \ v_y \ v_z \ 0)$.
- Note that vectors are unaffected by translations, as desired.

Perspective projection

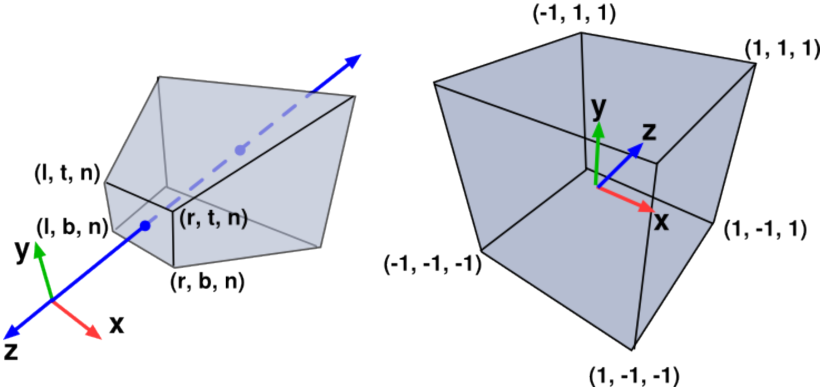
Requirements

- The projection has to project not only x and y, but also z.
- The projected z is used to compare the depth of different fragments and perform perspective correct interpolation.

Features

- The chosen projection matrix is a 4×4 matrix operating on homogeneous coordinates.
- It is non linear in $\mathbb{R}^3$, but linear in the affine space in which it operates.
- The projection matrix maps the view frustum volume in camera coordinates to a the normalized coordinates in clip space: $[-1..1] \times [-1..1] \times [-1..1]$ (the w is not considered here).

Perspective projection



Perspective projection matrix

OpenGL projection matrix

- The projection matrix defined by a view frustum r, l, t, b, n, f is:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

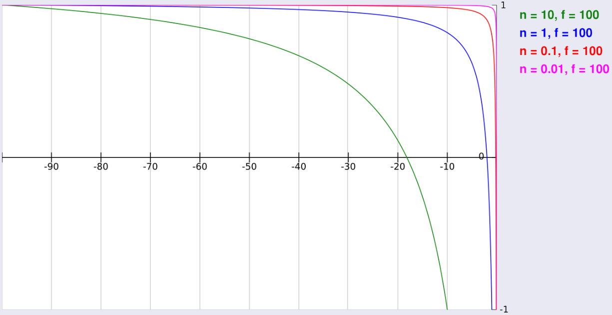
Find its derivation here: `http://www.songho.ca/opengl/gl_projectionmatrix.html`

Note that the camera is looking down negative $-z$, that means that the near plane is at $-n$ $(P(0\ 0\ \ -n\ 1)^t = (0\ 0\ \ -1\ 0)^t)$

- The final normalized device coordinates are obtained after perspective division (dividing x, y, z by w.

Z-transformation

- The projected value of z is not linear
- The closer *near* is to 0, the more the projected values are pushed towards 1:



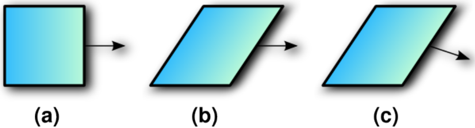
Graphs made with fooplot: `http://fooplot.com/`

- The z-values are normally discretized into 24-bits, so this can be a source of problems.

Other considerations

Transforming normal vectors

- When transforming a model, normals cannot be transformed with the same matrix than points.
- The inverse transposed is used instead (the prove uses the normal plane equation): $\boldsymbol{n'} = (M^{-1})^t \boldsymbol{n}$



From (a) to (b) the transformation applied is $S = \begin{pmatrix} 1 & \frac{2}{3} \\ 0 & 1 \end{pmatrix}$, however the correct normal in (c) requires $(S^{-1})^t = \begin{pmatrix} 1 & 0 \\ -\frac{2}{3} & 1 \end{pmatrix}$

Other considerations

Notation and composition

- Some APIs use row-major order for matrices and others use column-major order.
- In row-major order: $\boldsymbol{v' = M_1 M_2 v}$
- In column-major order the notation is: $\boldsymbol{v'^t = v^t {M_1}^t {M_2}^t}$
- In both cases, the first transformation applied is the closer to the vector.

Hidden surface removal

The cheapest pixel to render is that which is not rendered at all

Hidden surface removal techniques

- Object space
  - Back-face culling: Remove triangles that are facing backwards (determined by the vertex order during triangle setup)
  - View frustum culling: Use spatial partitions of the scene to determine which objects are inside the view frustum culling.
  - Occlusion culling: It can be a pure object space technique (e.g. portal culling) or combine image space with order space (hardware occlusion queries).
- Image space
  - Z-buffer algorithm: The algorithm used by GPUs to determine visibility during rasterization.

Lighting

Lighting

Lighting is the process of computing the final luminance of a pixel considering the (local or global) interactions between the light and the objects' materials.

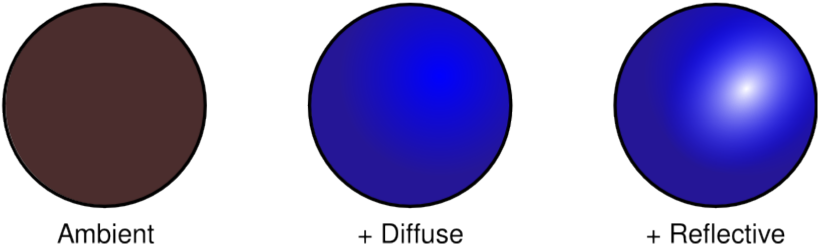
Lighting models vs shading models

In rasterization there is typically a distinction between lighting models and shading models.

- Lighting models: Specify how the light is reflected, emitted or transmitted by the surface (or volume).
- Shading models: Specify how the lighting model is applied (they have to do with sampling and interpolation).

Local illumination

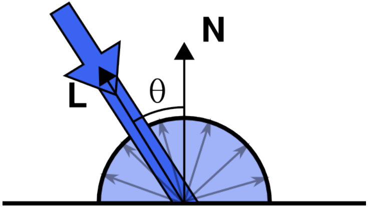
- Local illumination refers to the calculation of lighting based solely on the incoming light and properties at a point without considering the rest of the object.
- In real-time CG, the local illumination is typically the addition of three components:
    - Ambient
    - Diffuse
    - Specular



Lambertian reflectance

Definition, properties, formulation

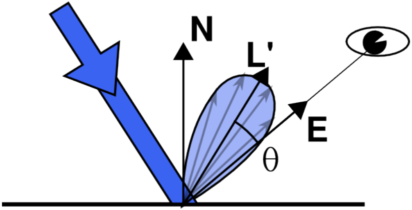
- Lambertian reflectance is the lighting model for perfectly diffusely reflecting surfaces (matte).
- In this model incoming light is reflected equally in all direction.
- The formulation is: $L_o = (\boldsymbol{N} \cdot \boldsymbol{L})K_d L_i$
- As a consequence of its definition, the lighting of lambertian diffuse surfaces is view independent.



Specular reflection

Phong specular lighting

- A simple model for glossy surfaces. The incoming light is reflected using the normal and depending on the angle between the reflected light and the eye vector, a portion of the pure light color is added to the pixel color.
- The formulation is: $L_o = (\boldsymbol{L'} \cdot \boldsymbol{E})^{\alpha} L_i$, where $\boldsymbol{L'} = \boldsymbol{L} + 2(\boldsymbol{N} \cdot \boldsymbol{L} - \boldsymbol{L})$



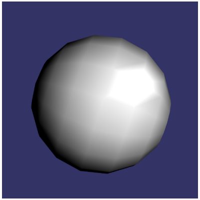
Shading models

Gouraud, Phong and deferred shading

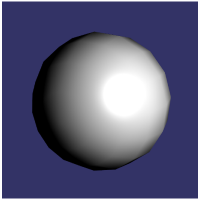
The most common shading models for local illumination (although Gouraud is becoming obsolete).

- Gouraud shading applies the lighting model at the vertices and interpolates colors at pixels.
- Phong shading applies the lighting model at each pixel. It interpolates the parameters of the lighting model.

Another alternative common in games, *deferred lighting*, is to write all lighting parameters to an output buffer and compute lighting as a post-processing step.



Gouraud

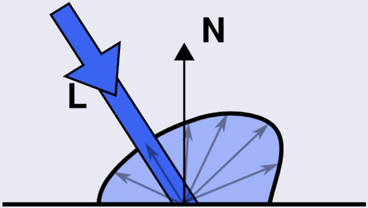


Phong

More advanced local illumination

BRDF

- *Bidirectional Reflectance Distribution Functions* are black box functions that for a pair of input and output directions return the ratio of the outgoing radiance to the incoming irradiance in those directions.
- These functions are used in physically correct rendering.



Global illumination

The rendering integral

Global illuminations methods try to approximate an equation known as rendering equation [Kajiya, 86]:

$$L_o(\boldsymbol{x},\boldsymbol{\omega}) = L_e(\boldsymbol{x},\boldsymbol{\omega}) + \int_\Omega f_r(\boldsymbol{x},\boldsymbol{\omega'},\boldsymbol{\omega})L_i(\boldsymbol{x},\boldsymbol{\omega})(\boldsymbol{\omega'}\cdot\boldsymbol{n})d\boldsymbol{\omega'}$$

Techniques

- Ray tracing.
- Radiostity.
- Ambient occlusion (interactive approximation).

More complex methods account for scattering, fluorescence, . . . , which are limitations of the formulation above.

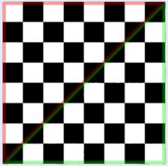
Textures

Textures

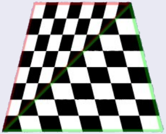
- A texture is uniform 1D, 2D or 3D grid with a value stored at each location of the grid.
- Textures are indexed by texture coordinates and values are accessed using some filter (nearest, bilinear, trilinear, . . .).
- They can be used to apply colors, retrieve properties used for shading, gradients, ...
- The resolution is each grid size is usually a power of 2 but non-power-of-two textures also supported by modern GPUs.
- Textures are a first class citizen in graphics APIs and GPUs. There is dedicated hardware for fetching, decompression and caching of texture memory as well as filtering.

Texture mapping

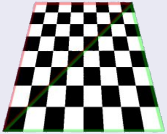
- Each vertices has a set of textures coordinates defined
- Texture coordinates are not linearly interpolated because it introduces a noticeable distortion in perspective projections.
- Instead the are interpolated using the inverse of the depth as an additional interpolation weight.
- Texture coordinates can also be transformed in the vertex processing stages.



Texture



Linear interpolation

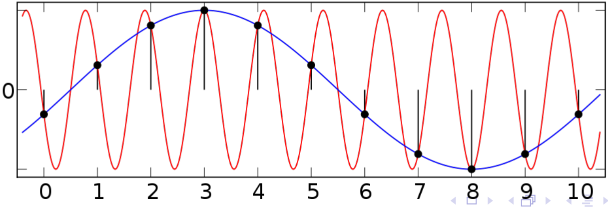


Perspective correct

Aliasing

What is aliasing

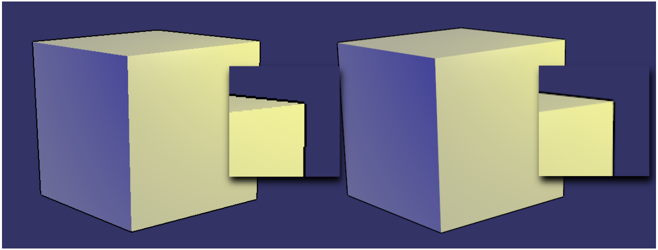
- Whenever a continuous function is sampled intro discrete samples aliasing is a problem.
- The Nyquist theorem states which is the maximum frequency that can be reconstructed for a given sampling frequency
- Aliasing occurs when the original signal contains frequencies above the Nyquist rate.
- Low-pass filtering is required.



Aliasing

Aliasing in CG

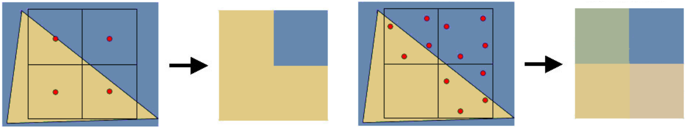
- In CG is common to find infinite frequency signals: creases and polygon borders.
- Texture sampling is also subject to aliasing problems: during texture mapping, textures are resampled to a different resolution than the original.



Polygon anti-aliasing

Techniques

- SSAA (super-sampling anti-aliasing): Render higher resolution then downsample (the brute-force approach).
- MSAA (multi-sample anti-aliasing): Use several depth and coverage samples per pixel but fewer color samples.
- MLAA (Morphological anti-aliasing): Post-processing anti-aliasing based on edge detection. Cheaper than SSAA and MSAA and slightly worse results than MSAA.



No AA compared to MSAA with 4 coverage samples per pixel

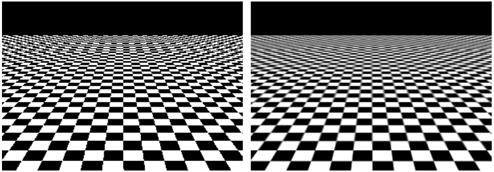
Texture sampling

Texture magnification

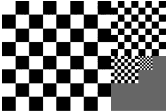
- Bilinear and trilinear filtering implemented in hardware.

Texture minimization

- Mip-maps: a texture depending where each level is a low-pass filtered and halved copy of the previous level. Trilinear sampling is performed between neighbour levels based on the distance.



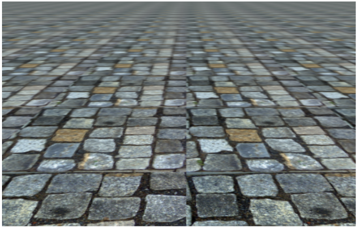
Mip-map texturing



Mip-map texture

Texture anisotropic sampling

- Mip-map filtering can cause blur when the texture is viewed at an oblique angle.
- Anisotropic filtering samples the mip-map considering the projection of the pixel on the texture, increasing the sharpness of the rendering.



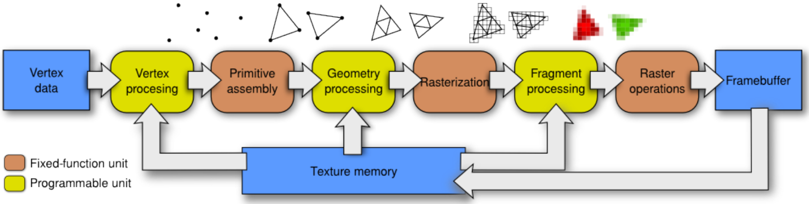
MIp-map trilinear filtering

Anisotropic filtering

1. Introduction to 3D Computer Graphics
    - Object modelling
    - Geometrical transformations
    - Lighting
    - Texturing and aliasing

2. Basic introduction to GLSL

The OpenGL graphics pipeline



GLSL

GLSL

- GLSL stands for OpenGL Shading Language.
- It is the language used to program the programmable stages of the OpenGL pipeline.

GLSL programs

A GLSL program is compiled into machine code that the GPU executes. The processing flow from source to execution is:

- A collection of plain text *shader* files is written.
- The files are passed to the driver, which compiles then in *shader objects*.
- The shader objects are linked into a *program*.
- The application sets the program inputs with the API.
- The program is uploaded and executed in the GPU.

Versions

- OpenGL 2.x: 1.1, 1.2
- OpenGL 3.x: **1.3**, **1.5**, 3.3
- OpenGL 4.x: 4.0, 4.1, 4.2

References

The best online documentation are the official specification and manual pages:

- http://www.opengl.org/sdk/docs/manglsl/
- http://www.opengl.org/documentation/glsl/

The *Orange Book* (OpenGL Shading Language 3^{rd} Ed., Randi J. Rost, Addison-Wesley) is also a good reference

Omissions

Features deliberately omitted in this slides: tessellation shaders, layouts, multiple buffer outputs, . . .

GLSL Types

Scalar types

The scalar types are the common one:

- `bool`, `int`, `unsigned int` and `float`

Newer versions also incorporate `double` and `half`.

Linear algebra types

GLSL has several built-in types for vectors and matrices:

- For vectors the types names are g`vec`n, where g can be: nothing for floats, `i` for integers, `u` for unsigned integers and `b` for booleans; n is one of 2, 3, or 4.
- Matrix types are all float: `mat2`, `mat3`, `mat4`, `mat3x3`, `mat2`, `mat2`
- Matrices are stored in column-major order, but the notation used for them is the usual one in math texts.

GLSL Types

Samplers

- Opaque handlers used in texture operations.
- Cannot be assigned, only passed as in parameters or declared globally.
- Initialized by the OpenGL API (host side).
- The most important sampler types are: `sampler1D`, `sampler2D`, `sampler3D`, `samplerRect`.

Structs

- Struct types can be declared using the same syntax than in `C`.

Arrays

- Arrays are declared and used like in `C`.
- There is no pointer type in GLSL (without extensions).

Qualifiers

- `const` Constant variables (initialized at compilation time or during function invocation).
- `in`: Read-only variables from the previous pipeline stage. Also used for read-only formal arguments in functions.
- `out`: Write variables from the previous pipeline stage. Also used for output formal arguments in functions.
- `inout`: Only for input output formal argument in function declarations.
- `uniform`: Input variable linked to a program through the API. Once a primitive is issued, the value does not change for it.

Variable declarations

Syntax and scope

- Variables are declared with the following syntax: *qualifier type identifier* [= *initializer*];
- Variable scope follows similar rules to `C`.
- Some qualifiers are not allowed in local variable declarations.

Example

```
const int x;
in vec3 lightPosition;
uniform sampler2D normalMap;
out vec4 color;
void main() {
 in vec3 int n = 10; // invalid
}
```

Shaders

Shader types

- Vertex shader: Takes a vertex and its attributes and outputs a vertex plus user defined output attributes.
- Geometry shader: Takes a single primitive (point, line, triangle) and outputs one or several primitives (point, line, line strip, triangle, triangle strip). The maximum number of output primitive is fixed in the API.
- Fragment shaders: Takes a rasterized fragment and computes its final color, optionally the final fragment depth can be also output.

Shaders

Example (Vertex shader)

```
in vec2 texCoordIn;
out vec2 texCoord;
void main()
{
  gl_Position =
    gl_ProjectionMatrix * gl_ModelViewMatrix *
    gl_Vertex;
  texCoord = texCoordIn;
}
```

Shaders

Example (Vertex shader)

```
in vec2 texCoordIn;
out vec2 texCoord;
void main()
{
  gl_Position =
    gl_ProjectionMatrix * gl_ModelViewMatrix *
    gl_Vertex;
  texCoord = texCoordIn;
}
```

Example (Geometry shader)

```
in texCoordIn[];
out vec2 texCoordOut;
void main()
{
  gl_Position = gl_PositionIn[0];
  texCoordOut = texCoordIn[0];
  EmitVertex();
  gl_Position = gl_PositionIn[1];
  texCoordOut = texCoordIn[1];
  EmitVertex();
  gl_Position = gl_PositionIn[2];
  EmitVertex();
  texCoordOut = texCoordIn[2];
  gl_Position = gl_PositionIn[0];
  EmitVertex();
  EmitPrimitive();
}
```

Shaders

Example (Vertex shader)

```
in vec2 texCoordIn;
out vec2 texCoord;
void main()
{
  gl_Position =
    gl_ProjectionMatrix * gl_ModelViewMatrix *
    gl_Vertex;
  texCoord = texCoordIn;
}
```

Example (Fragment shader)

```
unfiform sampler2D tex;
in vec2 texCoordOut;
out vec4 color;
void main()
{
  color = texture(tex, texCoordOut);
  gl_FragDepth = gl_FragCoord.z;
}
```

Example (Geometry shader)

```
in texCoordIn[];
out vec2 texCoordOut;
void main()
{
  gl_Position = gl_PositionIn[0];
  texCoordOut = texCoordIn[0];
  EmitVertex();
  gl_Position = gl_PositionIn[1];
  texCoordOut = texCoordIn[1];
  EmitVertex();
  gl_Position = gl_PositionIn[2];
  EmitVertex();
  texCoordOut = texCoordIn[2];
  gl_Position = gl_PositionIn[0];
  EmitVertex();
  EmitPrimitive();
}
```

Functions

Declaration and definition

- Functions can be declared as in `C`.
- With the addition of the `in`, `out` and `inout` qualifiers.
- Parameters without qualifier are considered.
- Recursion is NOT supported.
- Functions can be declared in one compilation unit and defined somewhere else.

Example

```
vec4 transform(vec4 v, out vec3 eye)
{
  vec4 w = gl_ModelViewMatrix * v;
  eye = -w.xyz;;
  return gl_ProjectionMatrix * w;
}
```

Control flow

Control flow

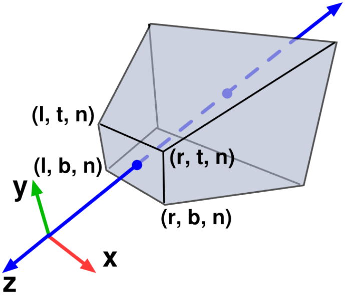
- GLSL supports almost all `C` control flow statements: `if...else`, `for`, `while`, `do...while`, `switch`.
- Normal flow can be interrupted with `break`, `continue` and `return`.

Discarding fragments

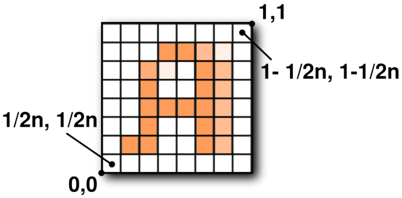
- The special single keyword statement `discard` can be used to stop processing in (and only in) fragment shaders.
- It affects a single fragment.
- The implementation must guarantee that the fragment shader execution has no side effects (no write to the framebuffer must occur).

Coordinates systems

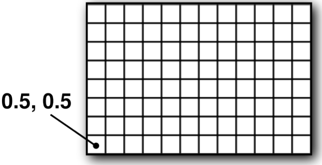
Camera coordinates



Texture coordinates



Viewport coordinates



Built-in variables

Global (all deprecated)

- `gl_ModelViewMatrix`: The local to camera coordinates transform matrix.
- `gl_ProjectionMatrix`: The projection matrix. transform matrix.
- `gl_NormalMatrix`: The local to camera matrix to use for normal vectors
- The inverse matrices are also available.

Vertex shaders

- `gl_Position`: Output variable that must be written.

Built-in variables

Geometry shaders

- `gl_PositionIn`: Input variable with the values output by the vertex shader for this primitive's vertices.
- `gl_Position`: Output variable that must be written.

Fragment shaders

- `gl_FragCoord`: Input variable with the x, y window coordinate and depth
- `gl_FragDepth`: Output variable to override the default depth calculation.
- `gl_FragColor`: The output color (*deprecated*).

Built-in functions

With vectors

- Scalar and vector products: `float dot(gvec, gvec)`, `vec3 cross(vec3, vec)`
- Length and normalization: `gvec normalize(gvec)`, `float length(gvec)`
- Reflection of a vector on the plane defined a normal vector: `gvec reflect(gvec ray, gvec n)`
- Elements access: `vec3 v`, `v[0] == v.r == v.x`
- Swizzling: `vec4 v`, `v.xyz`, `v.zx`, `v.rbg`
- Initializers: like a constructor call in `C++`. Can be used as declaration initializers or literals.
  - `vec3 p = vec3(1, 2, 3);`
  - `vec3 p = vec3(0.0);`
  - `vec4 q = vec4(p, 1.0);`

Built-in functions

With textures

- g`vec`x texture(*sampler*, `vec`x);

Geometry shaders

- `EmitVertex()`: pushes a new vertex into the primitive under constructions. All out variables must have been written before the vertex is emitted.
- `EmitPrimitive()`: finalizes the primitive under construction.

Other useful functions

- `min`, `max`, `pow`, `abs`, . . .
- `cos`, `sin`, `tan`, . . .
- Check the online manual pages: `http://www.opengl.org/sdk/docs/manglsl/`