Cube Mapping & Particle Systems in GLSL

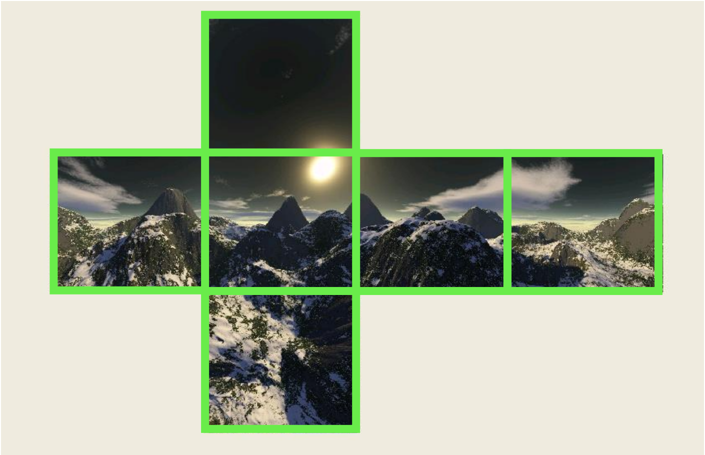
Part I: Cube Mapping

Overview

- Cube mapping
  - an environment mapping technique to simulate a shiny object reflecting its surrounding environment
  - Developed based on the assumption that the object's environment is relatively distant from the object
- A chrome-like appearance of an object can be achieved
  - by mapping the object with a **cube map texture** that encodes the object's environment

Cube Map Texture

- An omni-directional image of an environment
  - Consists of SIX images corresponding to the six faces of a cube, representing the views to the left, right, top, bottom, front and back of the environment
- Sampled using a 3D direction



How to Implement Cube Mapping

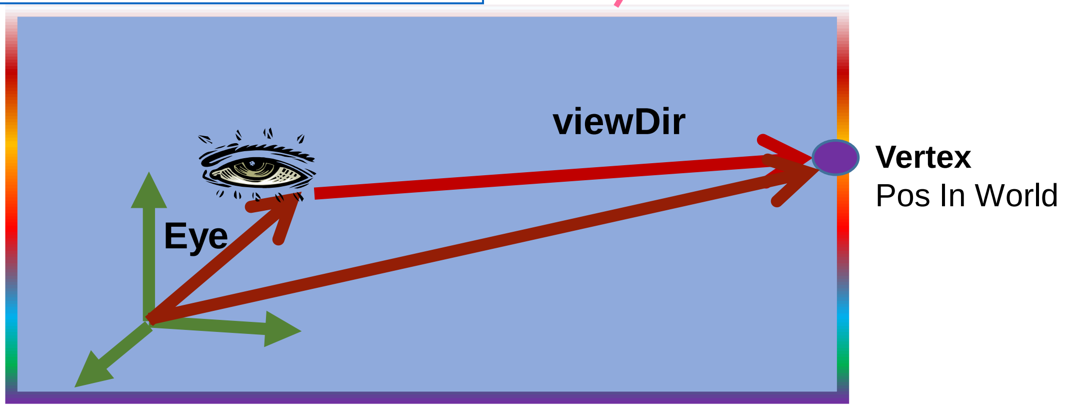
- Typically implemented in at least two passes:
  - Pass 1: Draw the environment
  - Pass 2: Draw the cube mapped object
- To draw the environment
  - Load a geometric object to model the environment
    - such as a **cube**, a **sphere**
  - Regard the geometric model as being very big and image its surface is distant from the viewer.
    - In this case, we can safely regard the view position is at the coordinate origin and just use the vertex position of the surface mesh as the view direction
  - depth test needs to be disabled

Draw Environment

Vertex Shader

```
uniform vec4 EyePosition;
varying vec3 viewDir;
void main(void)
{
    viewDir = gl_Vertex.xyz;
    vec3 PosInWorld = EyePosition.xyz + viewDir ;
    gl_Position = gl_ModelViewProjectionMatrix * vec4(PosInWorld , 1.0);
}
```

Environment mesh



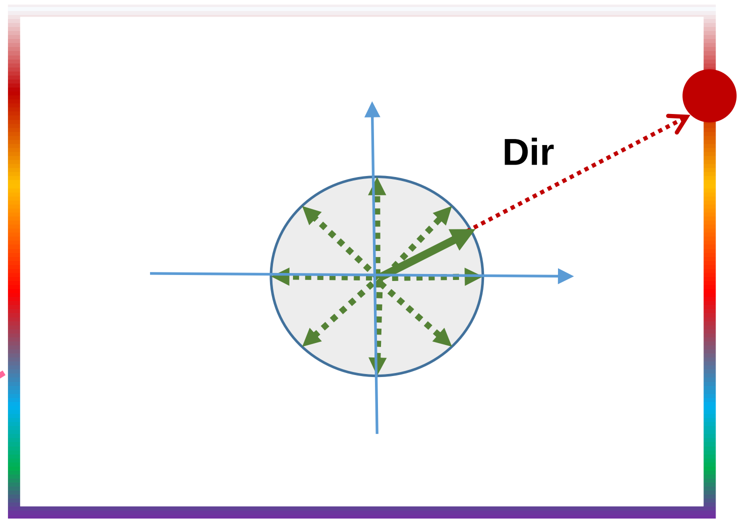
Draw Environment

Fragment Shader

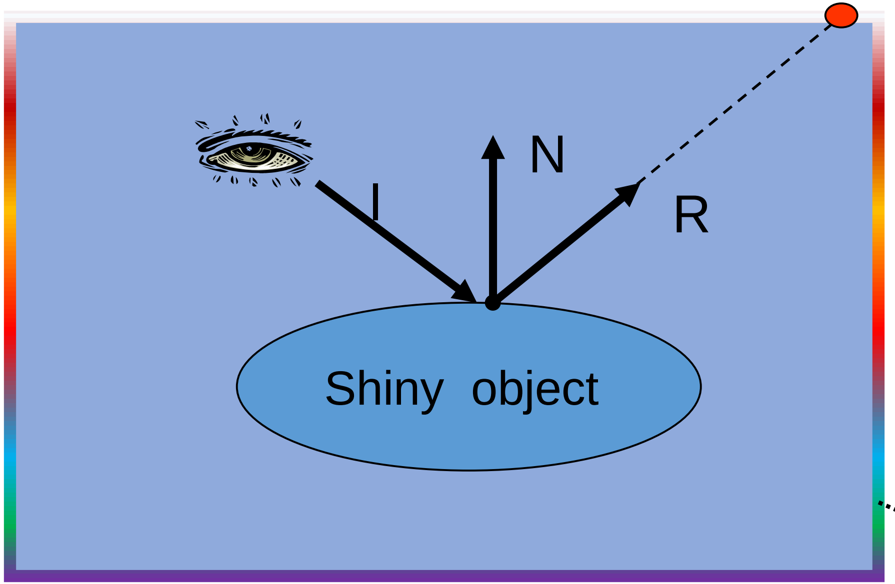
```
… …
uniform  samplerCube  skyBox;
varying vec3 viewDir;
void main(void)
{
   gl_FragColor = textureCube( skyBox, viewDir);
}
```

- Each direction points to a position on the cube map
- The pixel colour at the position can be obtained using GLSL function:
  textureCube(EnvironmentMap, **Dir**);

Cube map



Draw a Reflective Object



Calculate the reflection direction R:

R=I-2(N•I)N

Can be found using the GLSL function:

R=reflect (I, N)

Environment texture

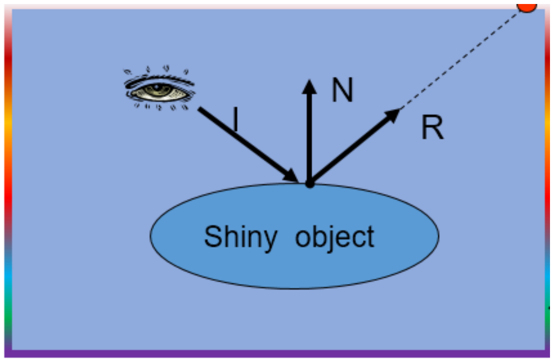
Vertex Shader

- Must pass information to fragment shader for computing the reflected view direction
- Two vectors are needed:

```
varying vec3 vNormal;            //normal vector
varying vec3 vView;              //view direction
```

- Compute normal and view vectors:

```
void main(void)
{
    … …
    vNormal = gl_Normal;
    vView = EyePosition.xyz – inPos.xyz;
    … …
}
```



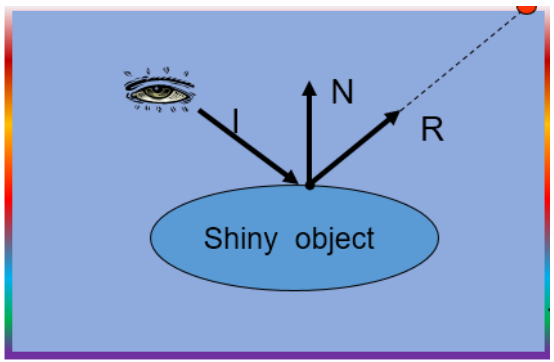
Fragment Shader

uniform **samplerCube** Environment;

varying vec3 vNormal;

varying vec3 vView;

… …



void main( ) {

… …

vec3 normal = normalize(vNormal);

vec3 V = normalize(vView);

// Find the reflected view direction

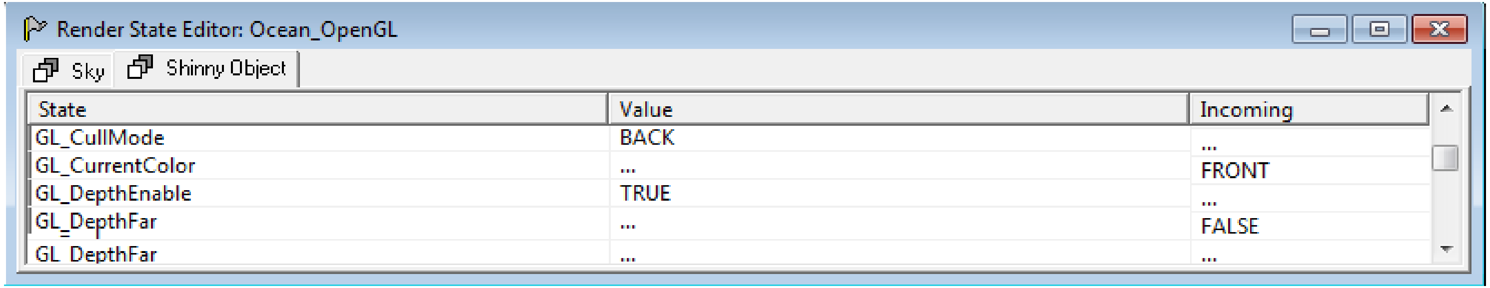
vec3 reflVec = reflect(-V, normal);

vec4 reflCol = textureCube(Environment, reflVec);

… …

}

Set the Render States Properly

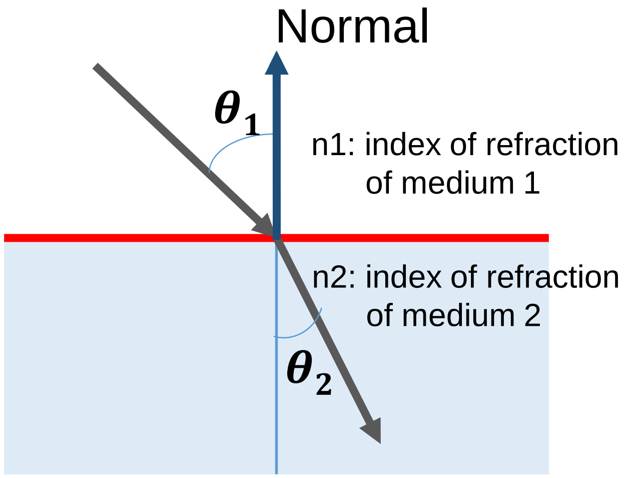


Light Refraction

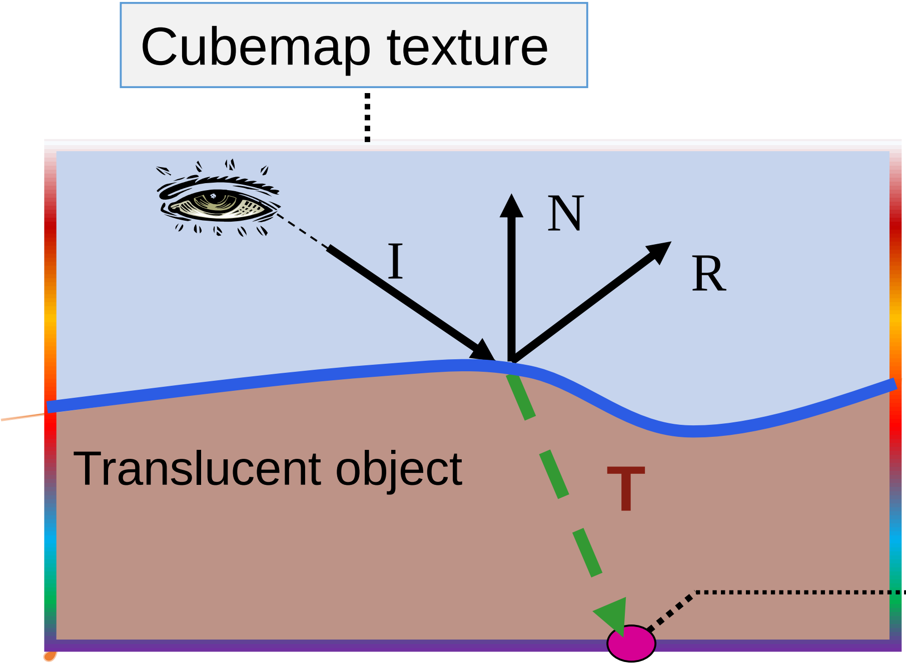
- A phenomenon of light
  - Describe the fact that the light direction will be changed when it passes from one medium to another
  - The degree of change depending on the type of the two media at the interface
  - Can be described by the **Snell's law:**

$$\frac{\boldsymbol{sin(\theta_2)}}{\boldsymbol{sin(\theta_1)}} = \frac{\boldsymbol{n_1}}{\boldsymbol{n_2}}$$

where $\frac{n_1}{n_2}$ is called **ratio of indices of refraction**



Refractive Environment Mapping



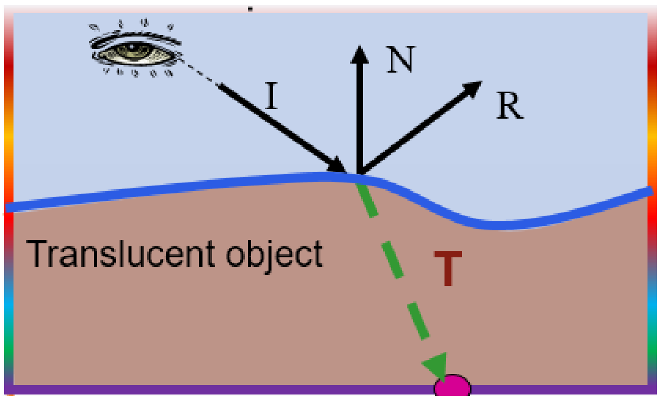
Refract direction can be found using the GLSL function:

T=refract (I, N, IndexRatio)

Texture colour corresponding to **T**:

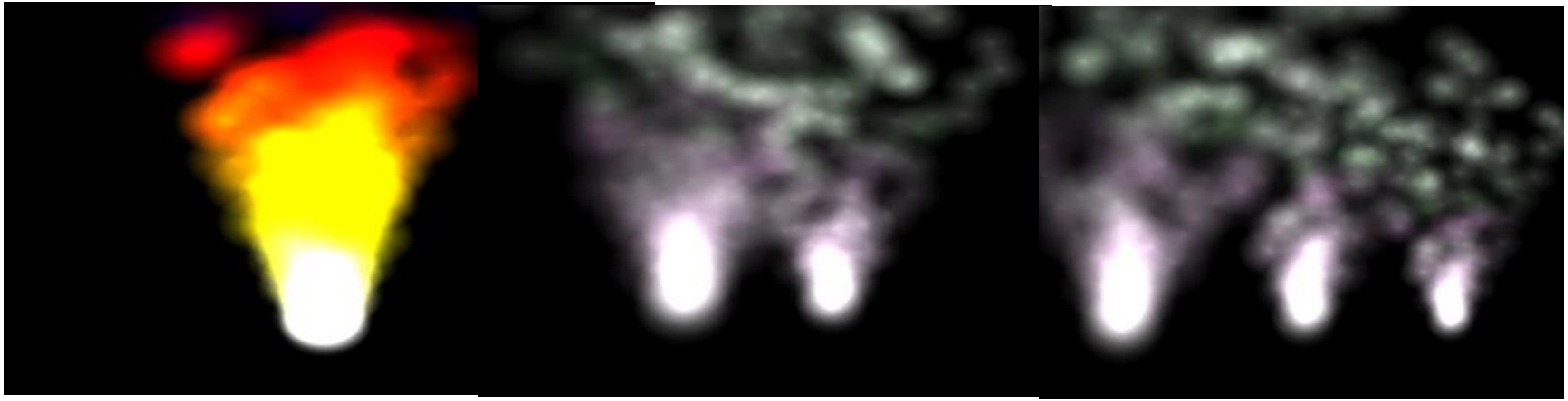
textureCube(EnvironmentMap, T);

Fragment Shader




```
… …
void main( ) {
  … …
   // Find the refraction
     vec3 refrVec =  refract(-I, normal, refractRatio);
     vec4 reflCol = textureCube( Environment, refrVec);
  … …
}
```

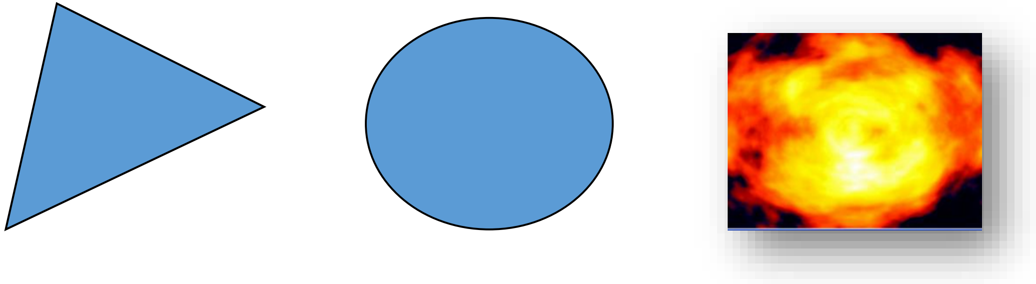
Implementation



Part II: Particle Systems

What Is a Particle

- A dynamically moving element
- Attributes of a single particle may include:
  - Shape
  - Texture
  - Position
  - Velocity
  - Mass
  - Life span



Modeling the Behavior of a Single Particle

- All attributes of the particle are functions of time
  - **Position**
    - move along certain path
    - Determined by all the forces acting on the particle
      - Force →Acceleration → Velocity → Position
  - **Colour**
    - Change along time
    - Depending the effects to be created
  - … …

Create Particles

- Create an array of simple geometric objects
  - In RenderMonkey, an array of quads can be created by loading the model QuadArray.3ds
    - QuadArray.3ds consists of a hundred quads, each of which is a [-1,1]x[-1,1] quad. The quads are differentiated by their z-value, ranging from 0 to 1.

  - You can use the positions of particles as parameters to specify how the particles are to be distributed in the space

Billboarding

- Billboard the quads.
  - Set xy-plane of each quad such that it always faces the viewer
- Specify a quad according to view space
  - Find the directions of x-axis and y-axis of view space
    - Which can be found from the inverse of ModelView Matrix:

```
uniform mat4  inverseViewMatrix;
```

    - Then the directions of x-axis and y-axis of view space are:

```
vec3 ViewLeft = inverseViewMatrix[0].xyz;
vec3 ViewUp = inverseViewMatrix[1].xyz;
```

  - Reset quad vertex position:

```
vec3 Pos = gl_Vertex.x * ViewLeft  +  gl_Vertex.y * ViewUp ;
```

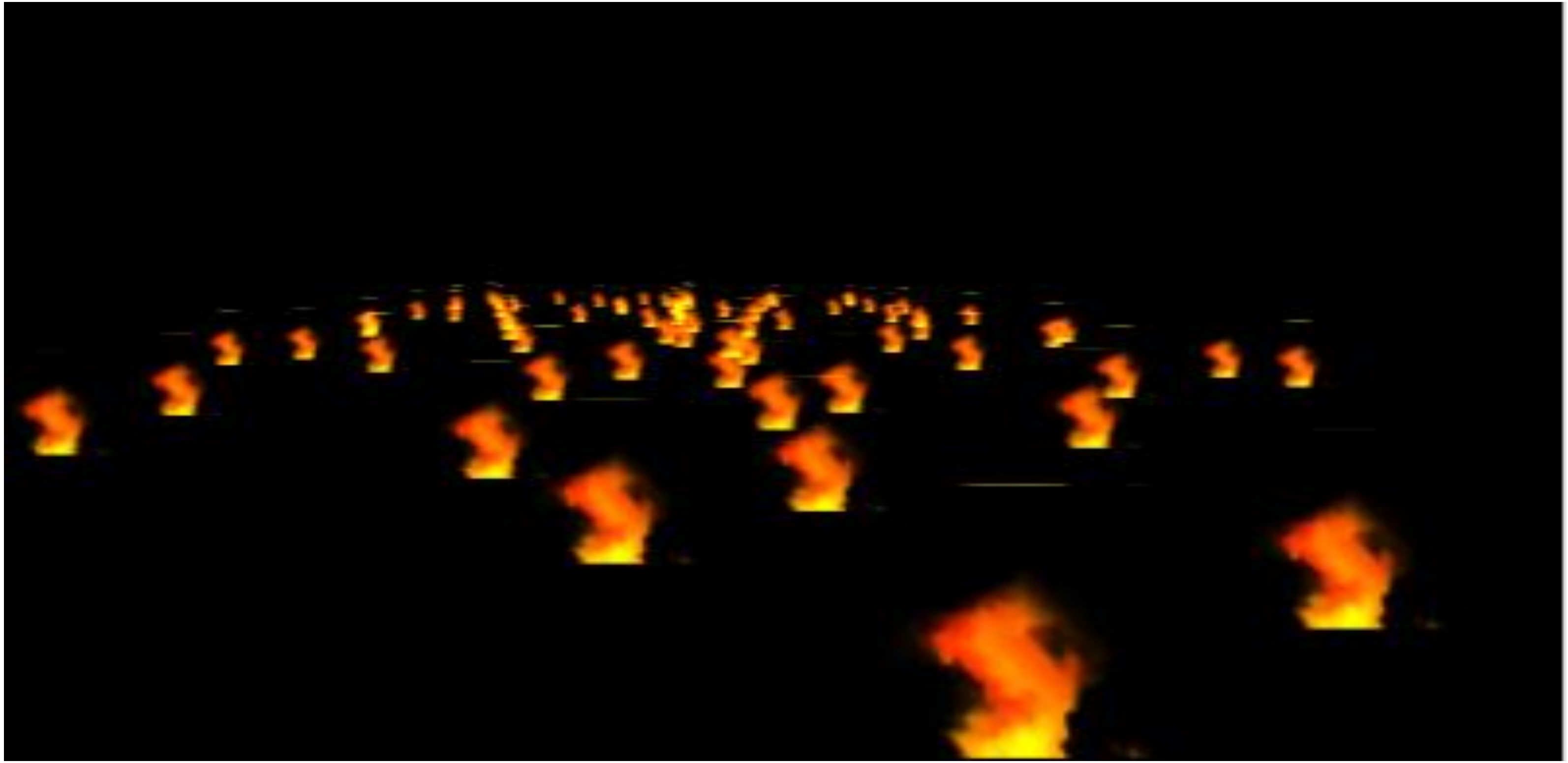
  - Reset quad size:

```
vec3 Pos *= particleSize;
```

  - Specify the quad position:

```
vec3 Pos += quadPos;
```

Example of Particle Systems



Example of Particle Systems

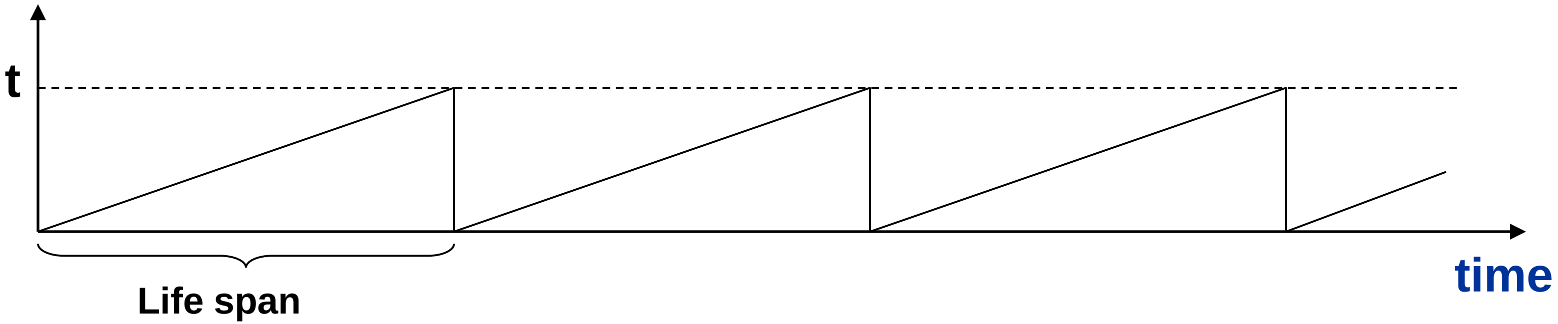
Fire Animation in Particle Systems

Modeling the Behavior of a Single Particle

- Life span

```
uniform float lifeSpan;
float t = mod (time, lifespan );
```

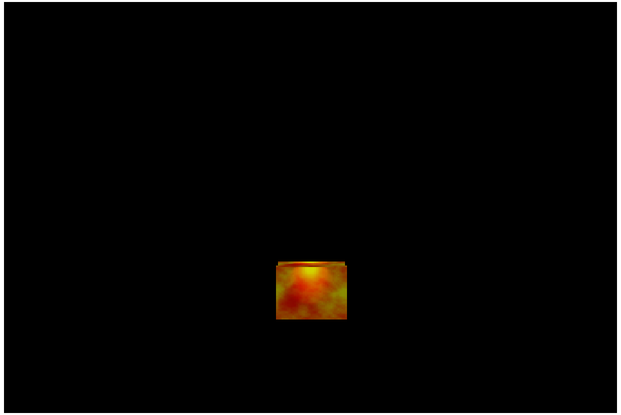
- The life span of a particle can also be modelled using function fract( )



Modeling the Behavior of a Single Particle

- Particle with constant velocity
  - Move along a straight line
  - Position of a particle can be calculated in the following way

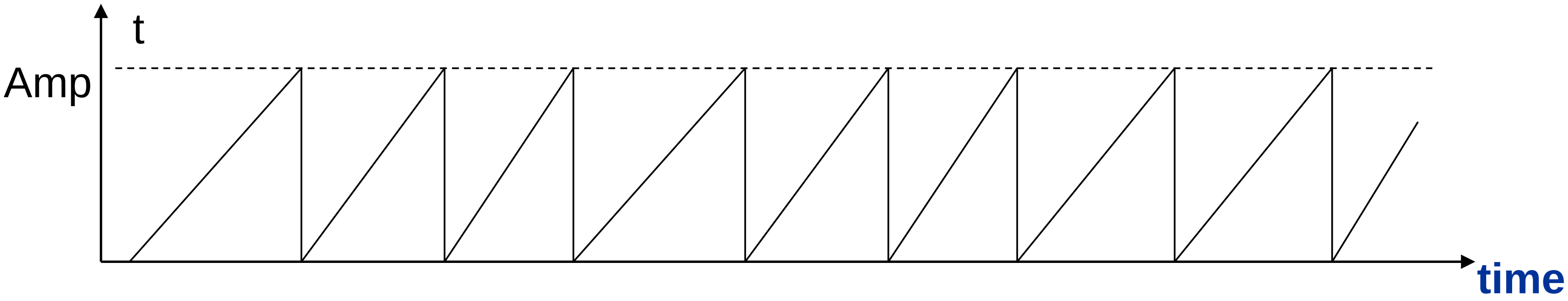
```
float t = mod (time, lifespan );
Pos.xyz  += velocity * t;
```



Modeling the Behavior of a Single Particle

- Control the lifetime and birth-death frequency of a particle

```
uniform float Amp;
uniform float freq;
float t = Amp*mod(freq*time, lifeSpan);
```



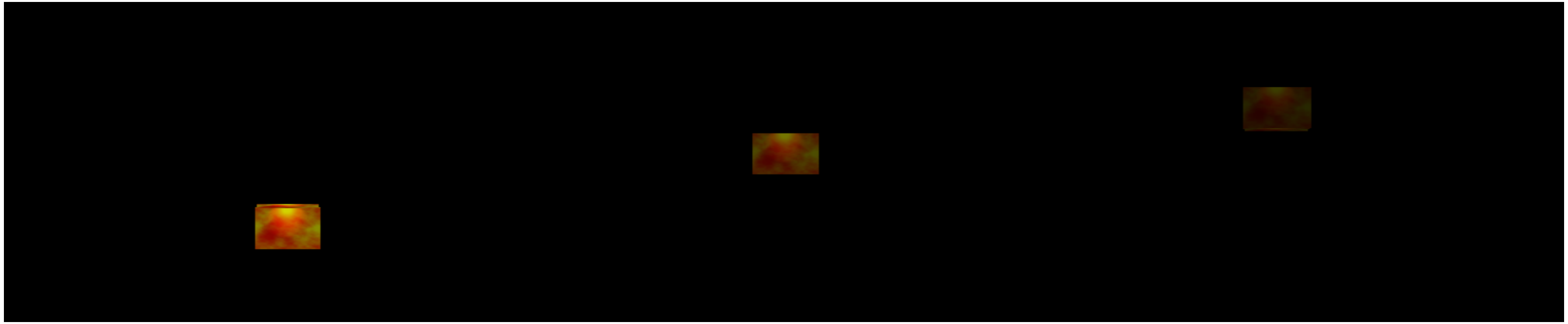
Modeling the Behavior of a Single Particle

- Update particle colour
  - Vertex shader:

    float fadingRate = (1 - t/ lifeSpan);

  - Pixel shader:

    return texture2D(baseMap, Texcoord)*fadingRate;

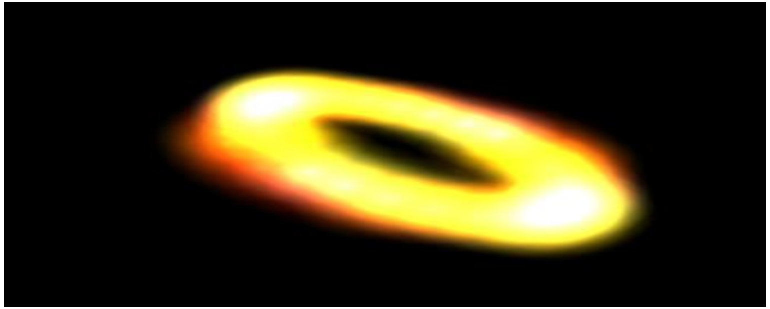


Modelling Fire Shape: Fixed Height

- Spread particles by regarding gl_Vertex.z as a parameter for specifying particle positions:
  - For example

```
Pos.x = particleSpread * cos(300 * gl_Vertex.z);
Pos.z = particleSpread * sin (300 * gl_Vertex.z);
Pos.y = particleSystemHeight;
```

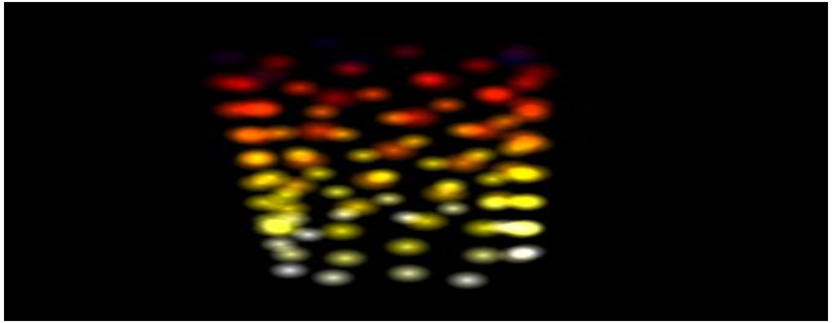
will spread the particles in the following way:



Modelling Fire Shape: Varying Height

Modify particle height varying with time:

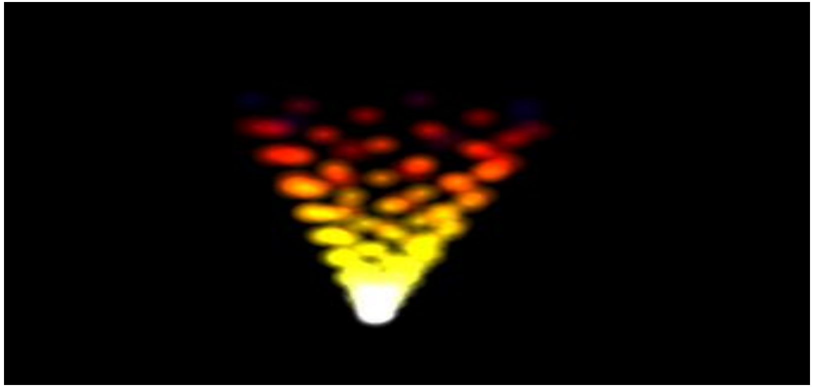
$$Pos.y = particleSystemHeight * t;$$



Modelling Fire Shape: Varying Radii

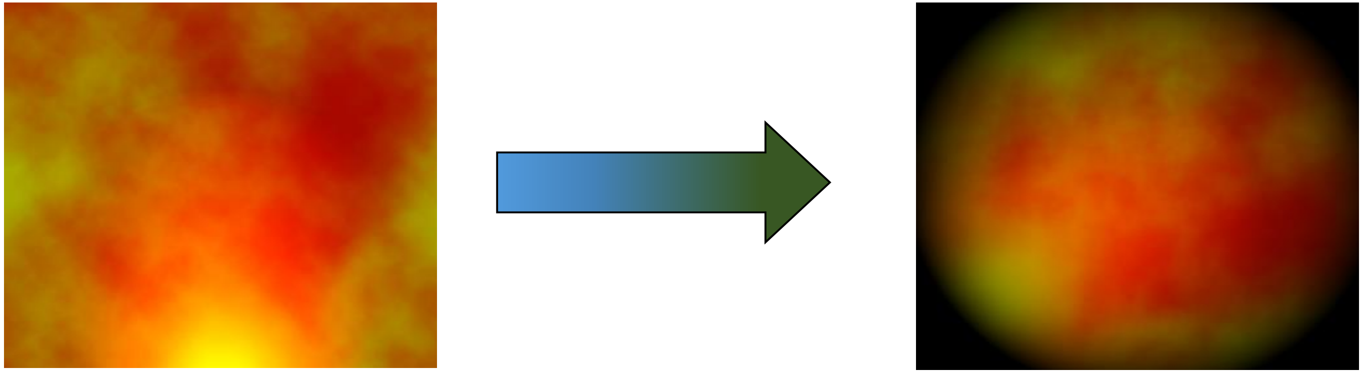
Modify particle spread using varying radii:

$$Pos.x = particleSpread * t * \cos(300 * gl\_Vertex.z);$$
$$Pos.z = particleSpread * t * \sin(300 * gl\_Vertex.z);$$
$$Pos.y = particleSystemHeight * t;$$



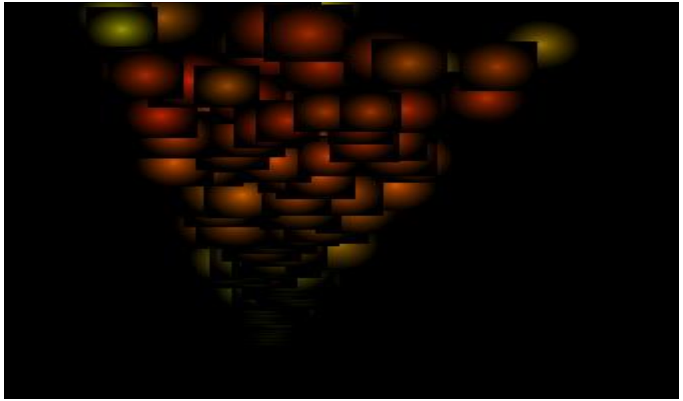
Modelling Fire Particle Colour

```
float x=TexCoord.x; float y=TexCoord.y;
float range= radius*radius - dot (TexCoord-0.5, TexCoord-0.5);
float shade = 2/(1+ exp(12* range));
return (1- shade)*tex2D( baseMap, Texcoord );
```

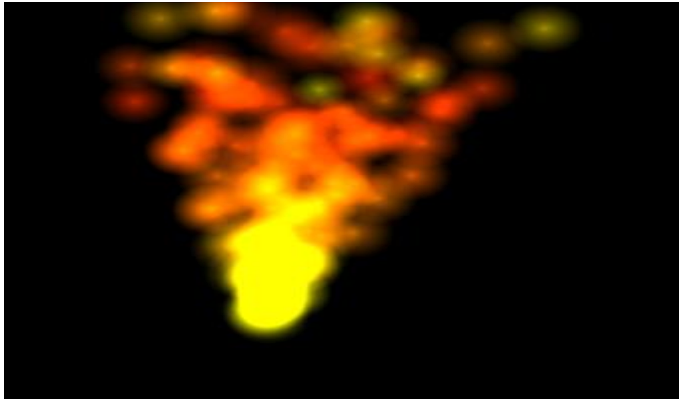


Particle Colour Blending

- Set GL_BlendEnable to be true to enable alpha transparency blending



Without alpha-blending



With alpha-blending

Questions?