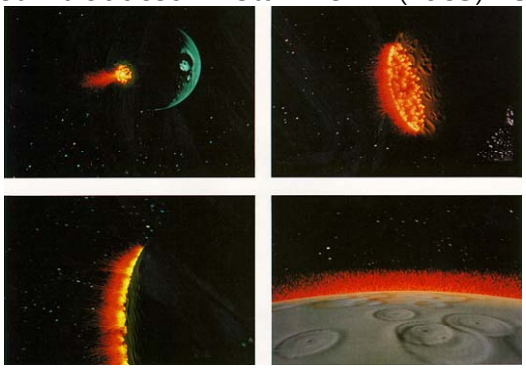
Particle Systems

305890
Spring 2012
6/11/2012
Kyoung Shin Park
kpark@dankook.ac.kr

Star Trek II (1983)

- Particle Systems can be utilized to simulate a wide range of phenomena such as fire, rain, smoke, explosions, and projectiles.
- It was first introduced in Star Trek II (1983) "Genesis Effect"

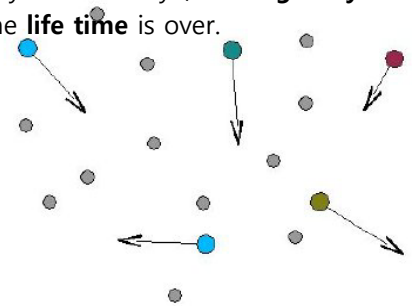


Particle Systems

- Particles (Point Sprite is removed from XNA4.0)
- Particle System Components
- Particle Systems
  - Snow/Rain
  - Explosions
  - Smoke
  - Fire

Particles

- Particle
  - A very small object that is usually modeled as a point mathematically.
  - Represented as a point with **position**, **velocity**, and **color**.
  - Each particle can be added in the particle system and moved in a physically realistic way (due to **gravity or wind**) and then died after the **life time** is over.



Particles

- Particle
  - Use a **Point Sprite** to represent a particle.
  - With point sprites (introduced in DirectX 8), graphics card only needs only one vertex for each particle.
    - Prior to point sprites, game developers had to create a rectangle (quad) and apply a texture to it for each particle.
  - Point sprites can be as small as one pixel to as large as we want
    - We can set each particle to be a different texture and/or size.
  - XNA automatically maps our texture to the vertex. It also makes sure that the texture is always facing the camera.
  - **XNA Game Studio 4.0 no longer supports point sprites http://blogs.msdn.com/b/shawnhar/archive/2010/03/22/point-sprites-in-xna-game-studio-4-0.aspx**

Particles

- Creating the ParticleVertex structure

```
public struct ParticleVertex {
    public Short2 Corner; // particle quad's corner
    public Vector3 Position;        // particle's starting position
    public Vector3 Velocity;        // particle's starting velocity
    public Color Random;// particle's color
    public float Time;      // the time at which the particle is created
    public static readonly VertexDeclaration VertexDeclaration = new
      VertexDeclaration (
        new VertexElement(0, VertexElementFormat.Short2,
                                  VertexElementUsage.Position, 0),
        new VertexElement(4, VertexElementFormat.Vector3,
                                  VertexElementUsage.Position, 1),
        new VertexElement(16, VertexElementFormat.Vector3,
                                  VertexElementUsage.Normal, 0),
    ... )
    `
```

Particle System

- Particle System Engine
  - Particle system class manages the multiple particles' creation, initialization, remove, update, and draw.
  - This class will be an abstract class that inherits from the DrawableGameComponent class.
  - Each individual particle system (smoke, explosion, fire, etc) will inherit from this base ParticleSystem class.

Particle System

- The particle system fields

```
// 전형적인 particle system을 일반화한 base class
pubic abstract class ParticleSystem :
    Microsoft.Xna.Framework.DrawableGameComponent {
    ParticleSettings settings = new ParticleSettings(); // particle settings
    ContentManager content;
    Effect particleEffect;
    EffectParameter effectViewParameter; // 중간생략 .....
    ParticleVertex[] particles;                    // particle array (circular queue)
    DynamicVertexBuffer vertexBuffer;          // for GPU
     IndexBuffer indexBuffer;
     int firstActiveParticle;     int firstNewParticle;
     int firstFreeParticle;       int firstRetiredParticle;
     float currentTime;
     int drawCounter;
     static Random random = new Random();
... }
```

Particle System

- Initialize the particle settings

```
// initialize particle settings
public override void Initialize()
{
      InitializeSettings(settings);
     // Allocate the particle array and fill in the corner (which never change).
     particles = new ParticleVertex[settings.MaxParticles * 4];
     for (int i = 0; i < settings.MaxParticles; i++) {
                particles[i * 4 + 0].Corner = new Short2(-1, -1);
                particles[i * 4 + 1].Corner = new Short2(1, -1);
                particles[i * 4 + 2].Corner = new Short2(1, 1);
                particles[i * 4 + 3].Corner = new Short2(-1, 1);
     }
     base.Initialize();
}
protected abstract void InitializeSettings(ParticleSettings settings);
```

Particle System

```
class FireParticleSystem : ParticleSystem  {
protected override void InitializeSettings(ParticleSettings settings) {
    settings.TextureName = "Particle₩₩fire";
    settings.MaxParticles = 2400;
    settings.Duration = TimeSpan.FromSeconds(2);
    settings.DurationRandomness = 1;
    settings.MinHorizontalVelocity = 0;    settings.MaxHorizontalVelocity = 15;
    settings.MinVerticalVelocity = -10;    settings.MaxVerticalVelocity = 10;
    settings.Gravity = new Vector3(0, 15, 0);
    settings.MinColor = new Color(255, 255, 255, 10);
    settings.MaxColor = new Color(255, 255, 255, 40);
    settings.MinStartSize = 5;
    settings.MaxStartSize = 10;
    settings.MinEndSize = 10;
    settings.MaxEndSize = 40;
    settings.BlendState = BlendState.Additive; // Use additive blending.
    }         }
```

Particle System

```
public override void Update(GameTime gameTime)  {
    currentTime += (float)gameTime.ElapsedGameTime.TotalSeconds;
    // if active particles have reached the end of their life, move to retired section
    RetireActiveParticles();
    // if retired particles have been kept long enough, move old particle to free
    FreeRetiredParticles();
    // If we let our timer go on increasing for ever, it would eventually
    // run out of floating point precision, at which point the particles
    // would render incorrectly. An easy way to prevent this is to notice
    // that the time value doesn't matter when no particles are being drawn,
    // so we can reset it back to zero any time the active queue is empty.
    if (firstActiveParticle == firstFreeParticle)
                currentTime = 0;
    if (firstRetiredParticle == firstActiveParticle)
                drawCounter = 0;
}
```

Particle System

```
public override void Draw(GameTime gameTime)  {
    if (firstActiveParticle != firstFreeParticle)  {      // if active particle, draw
          device.BlendState = settings.BlendState;
          device.DepthStencilState = DepthStencilState.DepthRead;
// 중간 생략.. effect.SetParameters

          // Set the particle vertex and index buffer.
          device.SetVertexBuffer(vertexBuffer);
          device.Indices = indexBuffer;
          // Activate the particle effect.
          foreach (EffectPass pass in particleEffect.CurrentTechnique.Passes)  {
               pass.Apply();
               if (firstActiveParticle < firstFreeParticle)  {
                    device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0,
                              firstActiveParticle * 4, (firstFreeParticle - firstActiveParticle) * 4,
                              firstActiveParticle * 6, (firstFreeParticle - firstActiveParticle) * 2);
               }
```

Particle System

```
                else    {
                    device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0,
                            firstActiveParticle * 4, (settings.MaxParticles - firstActiveParticle) *
                            firstActiveParticle * 6, (settings.MaxParticles - firstActiveParticle) *
                    if (firstFreeParticle > 0)     {
                        device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0,
                                                0, firstFreeParticle * 4,
                                                0, firstFreeParticle * 2);
                    }
                }
            }

            // Reset some of the renderstates that we changed,
            // so as not to mess up any other subsequent drawing.
            device.DepthStencilState = DepthStencilState.Default;
        }

        drawCounter++;
    }
```