COMPSCI 715 Part 2

Lecture 5 - ODE Solvers & Particle Systems cont..

Runge-Kutta Methods

- More accurate solution depends on higher-order estimates
- RK algorithm based on truncation of the taylor series:

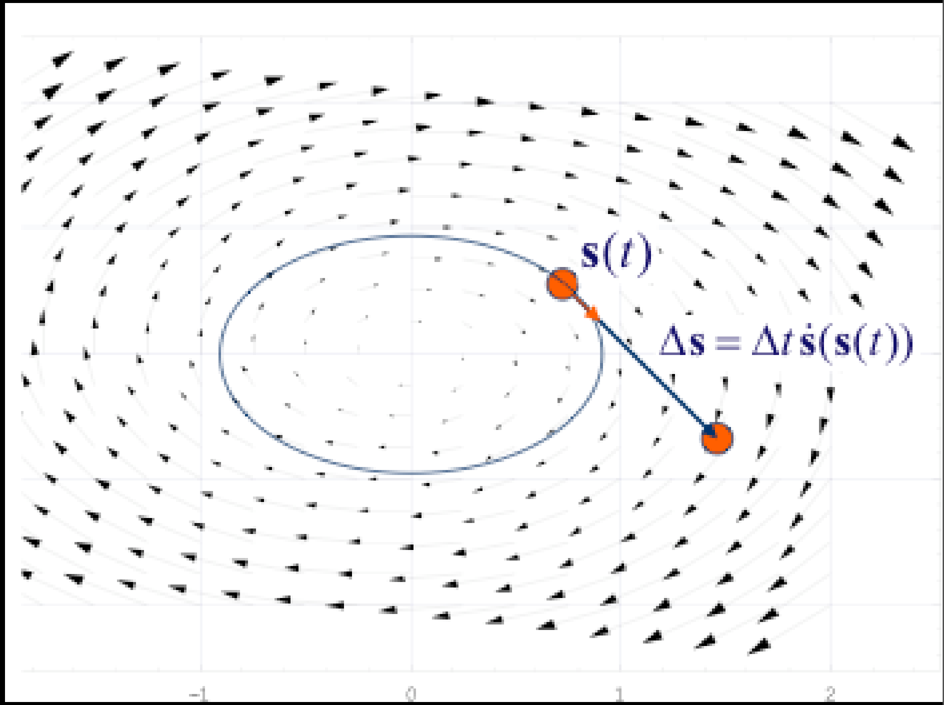
$$\underbrace{\mathbf{s}(t+\Delta t)=\mathbf{s}(t)+\Delta t\,\dot{\mathbf{s}}(t)}+\frac{\Delta t^2}{2!}\ddot{\mathbf{s}}(t)+\ldots+\frac{\Delta t^n}{n!}\frac{\partial^n\mathbf{s}(t)}{\partial t^n}+\ldots$$

$$\left[\dot{\mathbf{s}}(t),\ \ddot{\mathbf{s}}(t)\text{ are short for }\dot{\mathbf{s}}(\mathbf{s}(t)),\ \ddot{\mathbf{s}}(\mathbf{s}(t))\right]$$

Where RK(n) means first (n) terms taken
i.e Euler is RK1(in brace above)
Midpoint is RK2

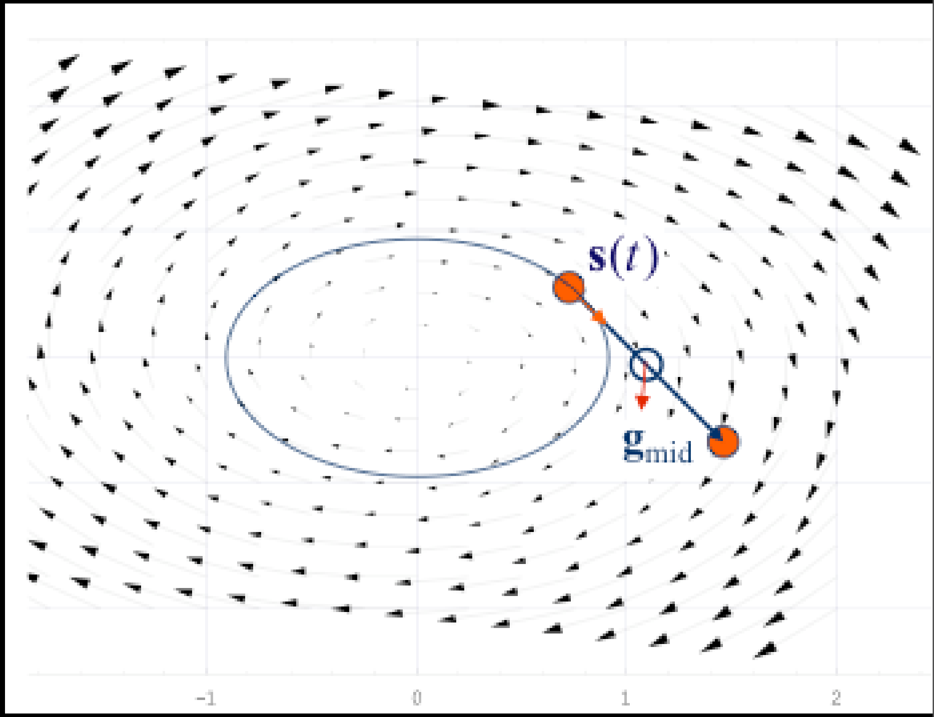
Midpoint method

- Compute Euler step using initial derivative
- Evaluate derivative at midpoint of step
- Return to start and use new derivative to take full step


$\mathbf{s}(t)$
$\Delta\mathbf{s} = \Delta t\,\dot{\mathbf{s}}(\mathbf{s}(t))$
-1
0
1
2

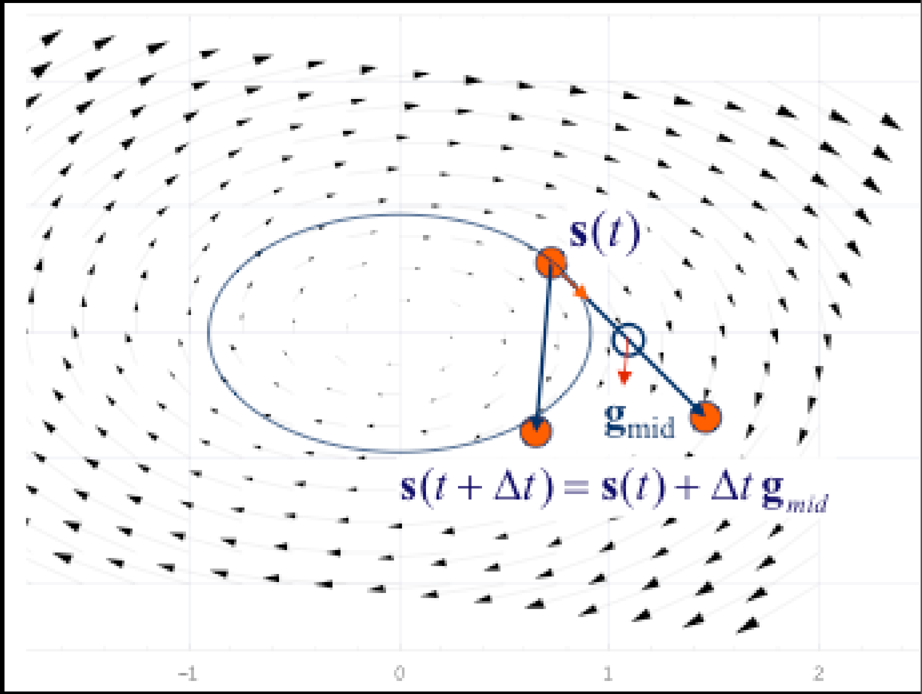
Midpoint method

- Compute Euler step using initial derivative
- Evaluate derivative at midpoint of step
- Return to start and use new derivative to take full step



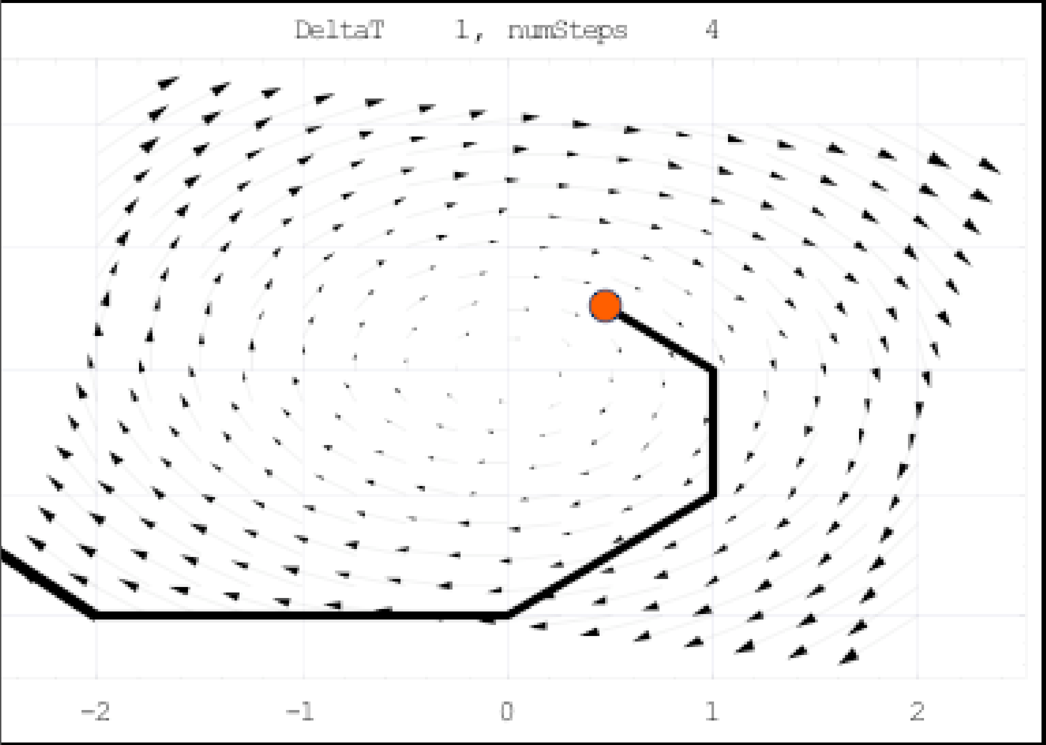
Midpoint method

- Compute Euler step using initial derivative
- Evaluate derivative at midpoint of step
- Return to start and use new derivative to take full step

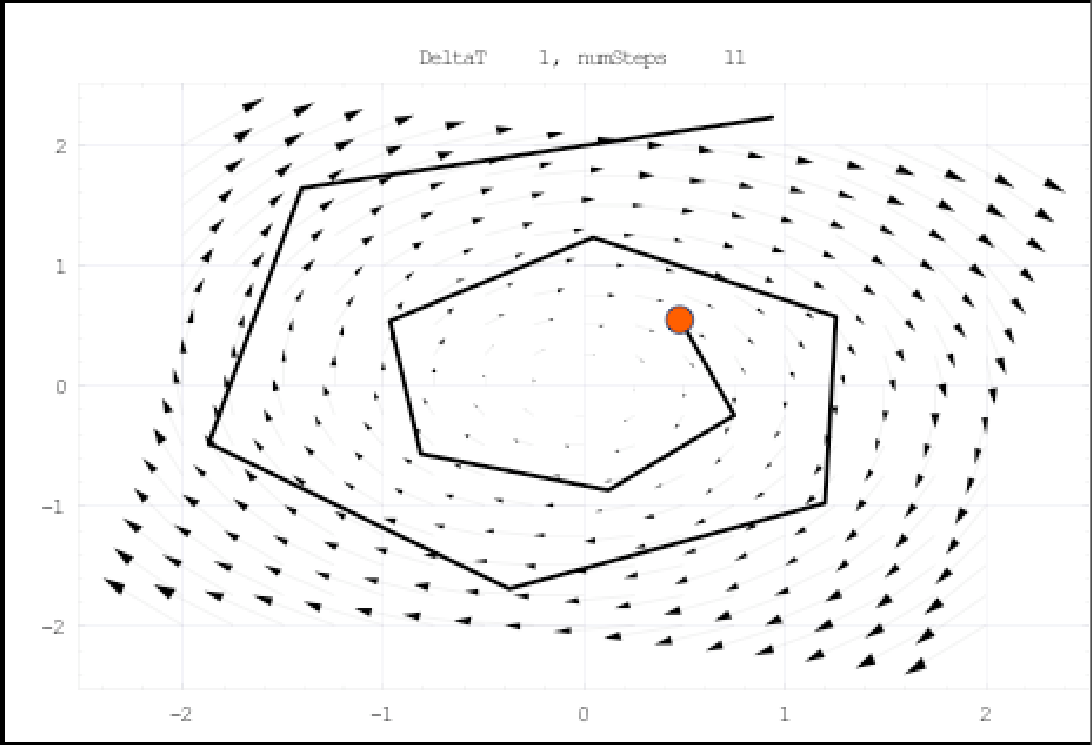


$\mathbf{s}(t)$
$\mathbf{g}_{\mathrm{mid}}$
$\mathbf{s}(t+\Delta t) = \mathbf{s}(t) + \Delta t\, \mathbf{g}_{mid}$
-1
0
1
2

Compare Midpoint with Euler



Euler Δt=1



Midpoint Δt=1

4th order Runge Kutta

"For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. ... Bulirsch-Stoeror predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields."

– Press et al, "Numerical Recipes"

RK4 Equations

$$\mathbf{k}_1 = \Delta t\,\dot{\mathbf{s}}(\mathbf{s}(t),t) \quad \text{(* Euler step } \Delta\mathbf{s} \text{ *)}$$

$$\mathbf{k}_2 = \Delta t\,\dot{\mathbf{s}}\left(\mathbf{s}(t)+\frac{\mathbf{k}_1}{2}, t+\frac{\Delta t}{2}\right) \quad \text{(* Midpoint step *)}$$

$$\mathbf{k}_3 = \Delta t\,\dot{\mathbf{s}}\left(\mathbf{s}(t)+\frac{\mathbf{k}_2}{2}, t+\frac{\Delta t}{2}\right) \quad \text{(* Refined midpoint step *)}$$

$$\mathbf{k}_4 = \Delta t\,\dot{\mathbf{s}}\left(\mathbf{s}(t)+\mathbf{k}_3, t+\Delta t\right) \quad \text{(* Refined refined step *)}$$

$$\mathbf{s}(t+\Delta t) = \mathbf{s}(t)+\frac{1}{6}\mathbf{k}_1+\frac{1}{3}\mathbf{k}_2+\frac{1}{3}\mathbf{k}_3+\frac{1}{6}\mathbf{k}_4 \quad \text{(* 4th order R-K result *)}$$

Back to Particle Systems

- Enhancements:
  - Point Sprites
  - Using the GPU

Point Sprites

- An advanced method of providing particle system geometry to the GPU
- Part of OpenGL 2.0
- Allows for hardware implementation of billboarding
- Specified as single point
  - Which is both good and bad

Point Sprites - Impl

- Use existing point functions for setting up
  - glPointSize()
  - glPointParameter for attenuation, max/min size and fading
- For automatic texture coordinate generation
  - glTexEnvf (GL_POINT_SPRITE,GL_COORD_REPLACE,GL_TRUE)

Point Sprite - Impl

- Enable with glEnable(GL_POINT_SPRITE)
- Draw with glBegin(GL_POINTS)

Demo

Particle Systems On the GPU

- Two types:
  - Stateless
  - State-preserving

Stateless

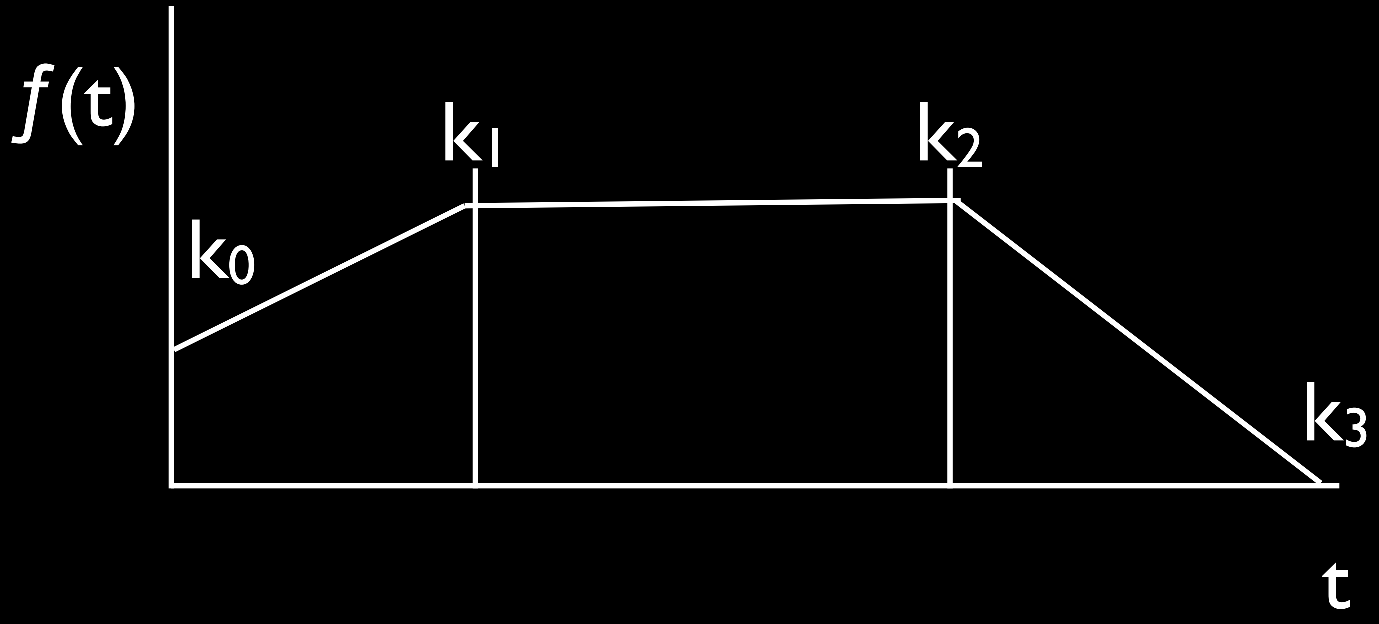
- Simplistic
- Use few parameters such as time of birth, acceleration, actual time and defined forces
- Implemented on the vertex shader

Stateless

1. Define colour as velocity(r,g,b) and time(a)
2. Place vertex at initial position
3. Use Euler to move vertex to current position
4. Draw point

Stateless

- Colour and opacity can be defined as linear segments with keyframes
- Linear interpolation between nearest



Stateless

Vertex Shader:

```
void main(void)
{
  vec4 vertex = gl_Vertex;

  float t = max(Time - gl_Color.a, 0.0);

  // modulo(a, b) = a - b * floor(a * (1 / b)).
  t = t - RepeatFactor * floor(t * (1.0 / RepeatFactor));

  vec3 velocity = Radius * (gl_Color.xyz - vec3(0.5));

  vertex += vec4(velocity * t, 0.0);
  vertex.y -= Acceleration * t * t;

  Color = vec4(gl_Color.rgb, 1.0 - t);
  gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

State-Preserving

- Closer to the CPU based implementation
- Lower CPU->GPU communication
- Stores all data on graphics card and updates within the fragment shader

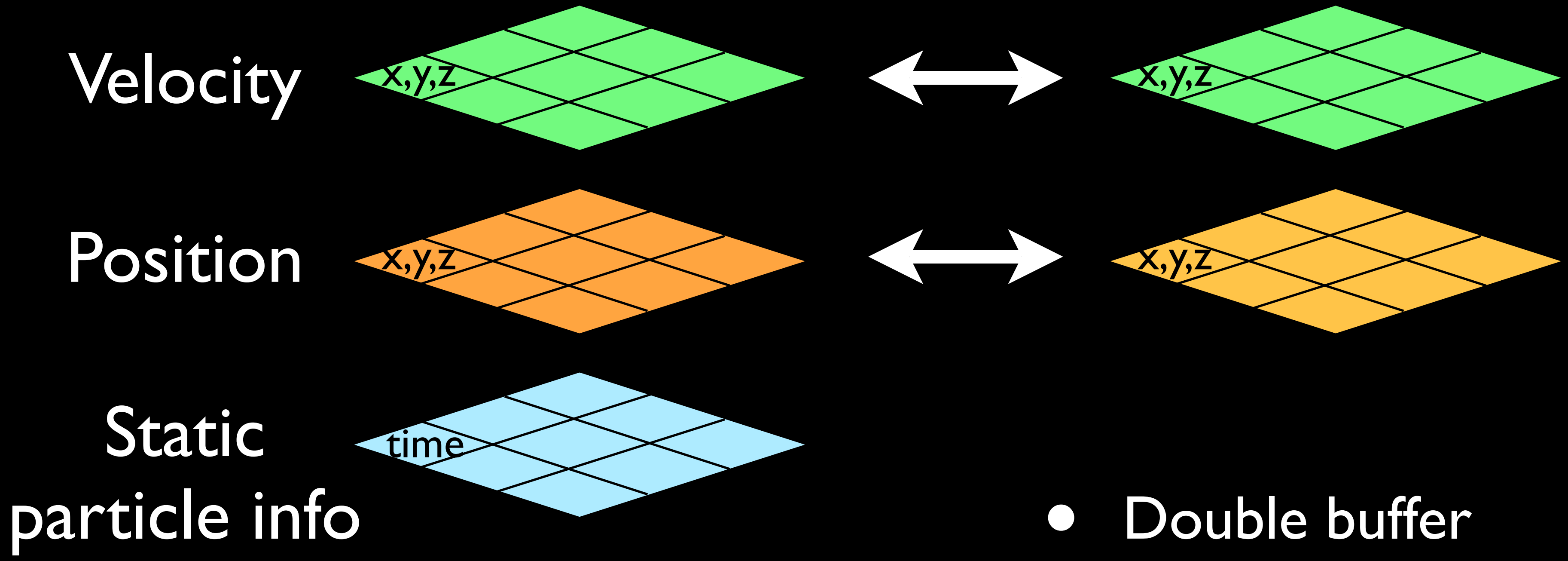
State Preserving

- Requires:
  - Floating-point pipeline (i.e. not [0,1])
  - Pixel-buffer objects
- Store particle state information on the GPU as textures (separate for position, velocity)

State Preserving

1. Particle Creation / Destruction
2. Pass 1: Update Particle Velocities
3. Pass 2: Update Particle Positions
4. Transfer pixel to vertex buffer
5. Render

State-Preserving



- Double buffer position and velocity textures to stop rendering over itself

Particle Creation

- Creation is a serial process so best for CPU
- Allocate based on next available position
- Render over top of previous information in position/velocity texture
- Dead particles are moved infinitely far away from view

Pass 1:Update Velocity

- Describe forces as one vector, sum of
  - Global Forces (gravity, wind)
  - Local distance based forces
  - Collision (limited to simple)
  - All implemented in the shader
- Update velocity using Euler

Pass 1:Update Velocity

Example Fragment Shader:

```
vec3 advanceVelocity(vec3 velocity, vec3 position, float timeStep){
   vec3 acceleration = vec3(0,-0.1,0); //gravity
   aceleration += localAttractor(position,Att0Pos,Att0Force,Att0Radius);
   aceleration += collision(position,velocity,Sphere0Pos,Sphere0Radius);
   velocity = veclocity + timeStep*acceleration;
   dampen(velocity);
   return velocity;
}

void main(void){
   vec3 velocity = vec3(texture2D(velocityTexture,  gl_TexCoord[0].st);
   vec3 position = vec3(texture2D(vositionTexture,  gl_TexCoord[0].st);
   gl_FragColor = vec4(advanceVelocity(velocity,position,TimeStep),0.0);
}
```

Update Position and transfer for render

- Use velocity texture to update position texture using Euler method
- Interpret position texture in vertex shader and reposition point sprites

Update Position

Example Fragment Shader:

```
void main(void){
   vec3 velocity = vec3(texture2D(velocityTexture,  gl_TexCoord[0].st);
   vec3 position = vec3(texture2D(vositionTexture,  gl_TexCoord[0].st);
   position = position + TimeStep * velocity; //Euler
   gl_FragColor = vec4(position,1);
}
```

Sources

- Latta, L (2004). Building a Million Particle System. GDC 2004
- Lobb, R (2003). Physically Based Animation Lecture Notes. COMPSCI715 2003