Graphics Programming Techniques

Dynamic GPU Particle System utilising Quaternion Mathematics

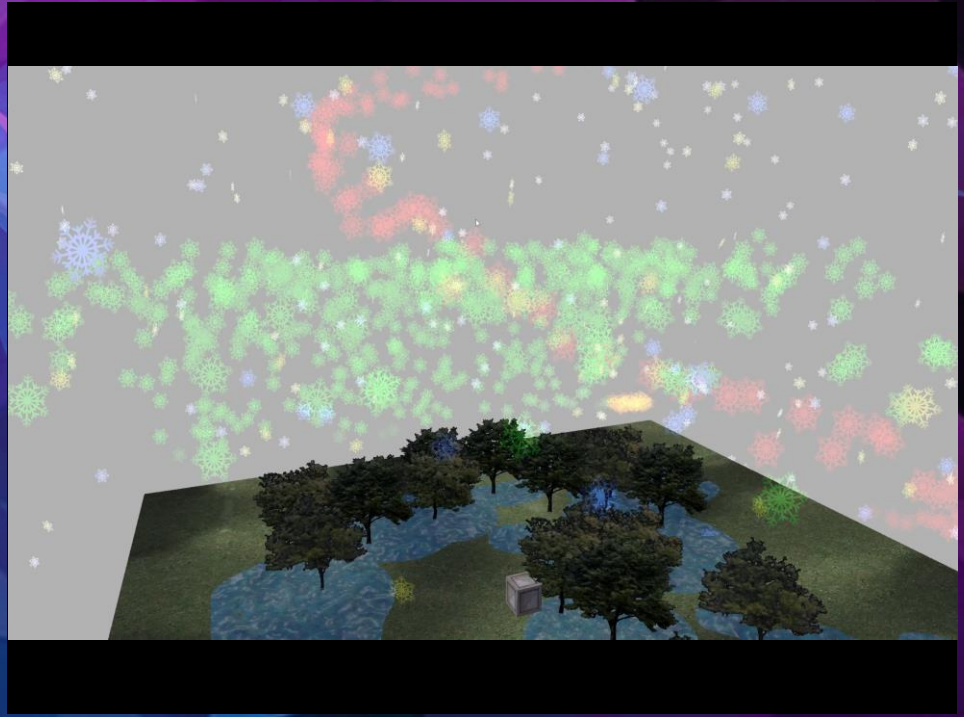
Joseph Mumford

Level 7 MComp | Sheffield Hallam University 2024

Overview

- This project entailed the creation of a dynamic particle system in DirectX 12, utilising the GPU to process and render high numbers of particles efficiently.

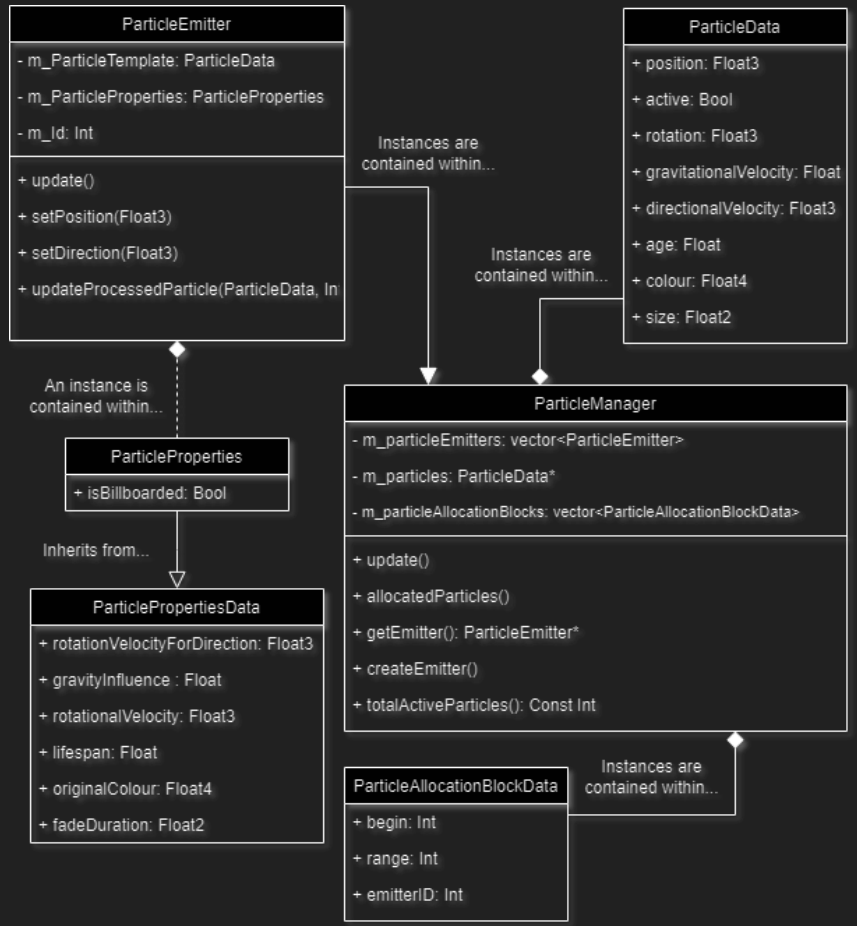
- “Dynamic” in this context refers to runtime creation, deletion, and variation of particles within this system; allowing various visual effects to be achieved.
  This video displays what the system is currently capable of; computing and rendering ~65,000 particles.



Particle System

- The implemented particle system involves encapsulating particle functionality between three main classes.
  These are Particles, Particle Emitters, and the Particle Manager.

- Particle Manager
  - A class that contains a pool of particle data as well as particle emitters.
- Particle Emitter
  - A class that activates, deactivates, and assigns properties to particle data.
- Particle(Data)
  - A struct containing data for GPU computation. Such as a particle's position and colour.

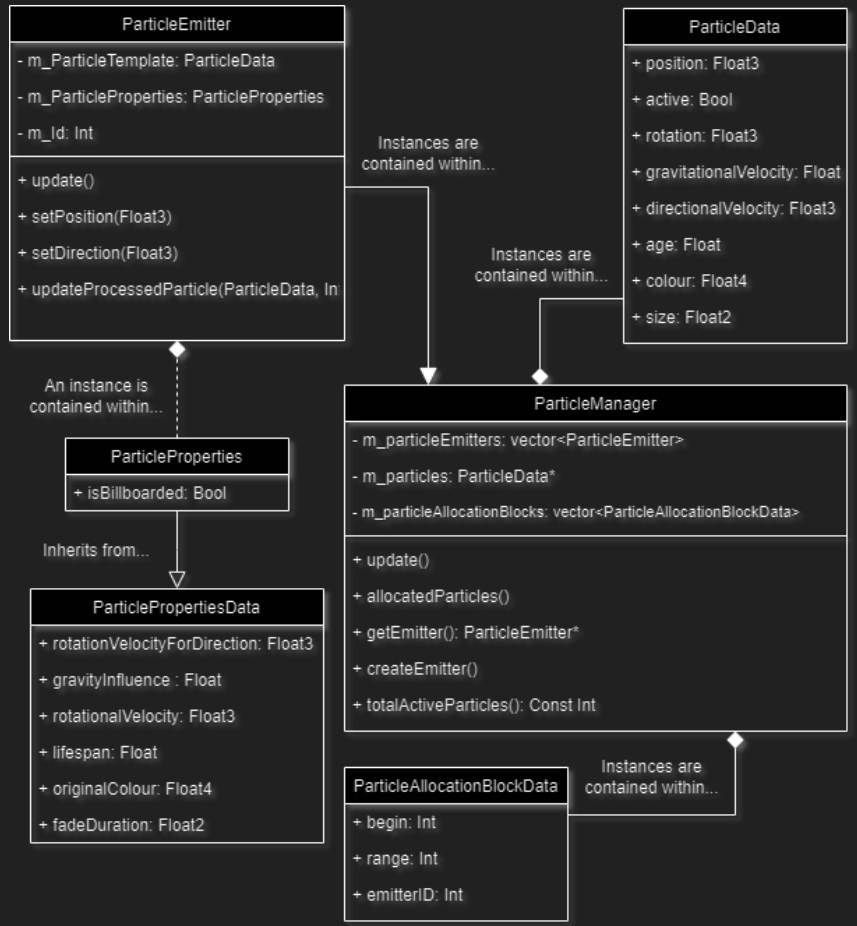


Particle System

- The system also includes some simple structs for particle allocation and behaviour.

- ParticleProperties & ParticlePropertiesData
  - Structs containing particle behaviour.
  - ParticlePropertiesData only contains data sent to the compute shader via a constant buffer.
  - ParticleProperties includes extra information not used for particle behaviour, such as billboarding.
  - This alleviates the ParticleData struct of unchanging variables that aren't updated within the compute shader.

- ParticleAllocationBlockData
  - A struct detailing a range of particles an emitter has exclusive access to. Utilised by the Particle Manager.



Shader Buffers

- In this system, both the graphics and compute shaders use input and constant buffers, but the latter also utilises output and readback buffers for retrieving processed data back into the particle manager.
- The compute shader utilises its constant buffer to receive ParticleProperties from Particle Emitters, which differs per emitter.

Graphics Shader Buffers (Vertex, Geometry, Pixel)

```
slotRootParameter[0].InitAsDescriptorTable(1, &texTable, D3D12_SHADER_VISIBILITY_PIXEL);
slotRootParameter[1].InitAsConstantBufferView(0);
slotRootParameter[2].InitAsConstantBufferView(1);
slotRootParameter[3].InitAsConstantBufferView(2);
```

Compute Shader Buffers

```
particleSlotRootParameter[0].InitAsConstants(1, 0);
particleSlotRootParameter[1].InitAsConstantBufferView(3);
particleSlotRootParameter[2].InitAsShaderResourceView(0);
particleSlotRootParameter[3].InitAsUnorderedAccessView(0);
```

- When dispatching to the compute shader, buffer data differs per emitter. Therefore, a dispatch call is made for each emitter, each with different input and constant buffer data.

```
// Update the Input Buffer
UpdateInputBuffer(emitter, cmdList);

// Update Constant Buffer
auto particleCB = mConstantBuffer.get();
particleCB->CopyData(0, emitter->getParticlePropertiesAsData());
```

```
// Map the data so we can read it on CPU.
ParticleData* mappedData = nullptr;
ThrowIfFailed(mReadBackBuffer->Map(0, nullptr, reinterpret_cast<void**>(&mappedData)));

// Read back the resulting particle data into the manager.
for (int i = 0; i < particleCount; ++i)
{
    emitter->updateProcessedParticle(ParticleData(mappedData[i]), i);
}
```

Particle Computation via a Compute Shader

- As part of the particle system, the compute shader is responsible for updating all particles dispatched to it, using the dispatched information.
  This includes updating positions, rotations, colour, opacity, travel direction, and velocity.
  Tasks that are infeasible on the CPU in high quantity.

- This enables the compute shader to achieve effects such as particles fading in and out, as well as changing colour or size over time; all done through use of the constant buffer and particle data. This varies the visual effects between particles and emitters.

```
cbuffer cbUpdateSettings
{
    // DeltaTime
    float gDeltaTime;
};
```

```
cbuffer cbParticleSettings : register(b3)
{
    float3   dDirRotVel;          // 96 bits:
    float    dGravityInfluence;   // 32 bits:

    float3   dRotVel;             // 96 bits:
    float    dLifespan;           // 32 bits:

    float4   dColorOriginal;      // 128 bits:

    float2   dFadeDuration;       // 64 bits:
};
```

```
struct ParticleData
{
    float3  dPosW;              // 96 bits:    World Position in XYZ
    bool    kActive;            // 32 bits:    Whether this particle is active or not.

    float3  dRotW;              // 96 bits:    XYZ Rotation as a Quaternion
    float   dGravityVelocity;   // 32 bits:    How much gravitational velocity is currently affecting this particle.

    float3  dDirVel;            // 96 bits:    Particle travelspeed and direction as a multiplication of a scaler and a unit vector.
    float   dAge;               // 32 bits:    The particle's age in seconds. This is incremented with gDeltaTime.

    float4  dColor;             // 128 bits:   RGB + Alpha / Transparency.

    float2  dSizeW;             // 64 bits:    Particle height and width
};
```

Vertex Data Construction + Instancing

- Once all active particles have been computed, their data is passed to the Graphics Shaders via a “ParticleVertex” struct.
  Therefore, particle data for all particles must first be moved from the Particle Manager into these structs.
  This includes colour, position, rotation, size, and whether the particle is billboarded; varying the particles, alongside the mathematics in the geometry shader.

- This is currently done on the CPU and, as a result, bottlenecks the system.
  Performing this on the GPU in future iterations would be more efficient, and solve this bottleneck.

- Once the particle vertices have been constructed, DrawInstance() is used to draw instances of particles with one draw call, rather than multiple.

Quaternions in the Geometry Shader

- Understanding of quaternions sourced from Frank Luna, among others.
  Introduction to 3D Game Programming with DirectX 12: Chapter 22 (2016)

- Quaternions are ordered 4-tuples, or a vector of 4 floating point numbers. They operate under the same principles of complex numbers, possessing “real” and “imaginary” components. For quaternions, the X, Y, and Z components are the latter, whereas their W component is the former.

- Complex numbers easily represent points and vectors due to how their real components multiply, add and subtract*. Quaternions expand this into 3D by possessing three imaginary components.

- Quaternions for spatial rotations can be constructed from a rotation direction and a magnitude (also referred to as the axis and angle respectively). For particles, we can pass in Euler angles and convert them into a quaternion for use in rotational transformations.

* $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$.
$(a, b) \pm (c, d) = (a \pm c, b \pm d)$.
$(a, b)(c, d) = (ac - bd, ad + bc)$.

[Code Flow | Non-Billboarded] Quaternions in the Geometry Shader

```
// Values required regardless of whether the particle is billboarded or not.
float   halfWidth   = 0.5f * gin[0].SizeW.x;
float   halfHeight  = 0.5f * gin[0].SizeW.y;
float4  v[4];

// Store the offsets of the vertices from the particle's center irrespective of rotation.
float3 o[4];
o[0] = float3(halfWidth, -halfHeight, 0);
o[1] = float3(halfWidth, halfHeight, 0);
o[2] = float3(-halfWidth, -halfHeight, 0);
o[3] = float3(-halfWidth, halfHeight, 0);

// Calculate the imaginary vector part and the real part for quaternion mathematics from
// the euler rotations in gin[0].RotW.xyz. The imaginary part is the direction of rotation,
// whereas the real part is the magnitude of rotation.
float3  qI   = normalize(gin[0].RotW.xyz);   // "Direction" of Rotation - normalize does what one would expect and returns a float3.
float   qR   = length(gin[0].RotW.xyz);      // "Magnitude" of Rotation - 'length' returns sqrt(x^2 + y^2 + z^2) as a float.

// Rotate the offsets of the vertices (float3 o[4]) around the world origin, then add the
// new offsets to the particle center (gin[0].PosW.xyz) to set the position of the newly built vertices.
v[0] = float4(gin[0].PosW.xyz + RotateVertexAroundPosition(o[0], qI, qR), 1.0f);
v[1] = float4(gin[0].PosW.xyz + RotateVertexAroundPosition(o[1], qI, qR), 1.0f);
v[2] = float4(gin[0].PosW.xyz + RotateVertexAroundPosition(o[2], qI, qR), 1.0f);
v[3] = float4(gin[0].PosW.xyz + RotateVertexAroundPosition(o[3], qI, qR), 1.0f);
```

The content of this function will be revealed in the coming slides. It does as described.

[Code Flow | Billboarded] Quaternions in the Geometry Shader

```
// Values required regardless of whether the particle is billboarded or not.
float   halfWidth   = 0.5f * gin[0].SizeW.x;
float   halfHeight  = 0.5f * gin[0].SizeW.y;
float4  v[4];
// Store the particle quad's halfWidth and halfHeight as an offset from a 2D world origin.
float2 vPoint[4];
vPoint[0] = float2(halfWidth, -halfHeight);
vPoint[1] = float2(halfWidth, halfHeight);
vPoint[2] = float2(-halfWidth, -halfHeight);
vPoint[3] = float2(-halfWidth, halfHeight);

// Rotate vPoints around the 2D world origin.
float s = sin(gin[0].RotW.z);
float c = cos(gin[0].RotW.z);
for (int i = 0; i < 4; ++i)
{
    float newX = (vPoint[i].x) * c - (vPoint[i].y * s);
    float newY = (vPoint[i].x) * s + (vPoint[i].y * c);
    vPoint[i] = float2(newX, newY);
}

// Build the new vertices with the width and height offset.
v[0] = float4(gin[0].PosW.xyz + (vPoint[0].x * right) + (vPoint[0].y * up), 1.0f);
v[1] = float4(gin[0].PosW.xyz + (vPoint[1].x * right) + (vPoint[1].y * up), 1.0f);
v[2] = float4(gin[0].PosW.xyz + (vPoint[2].x * right) + (vPoint[2].y * up), 1.0f);
v[3] = float4(gin[0].PosW.xyz + (vPoint[3].x * right) + (vPoint[3].y * up), 1.0f);
```

Rotation of the vertex position in 2-dimension space around a central point. This central point is around an abstract 2D world origin of 0,0 (x,y).

Quaternion Mathematics

- These are the quaternion functions used both in the geometry and compute shader.
- “Both”, because these function(s) can also be utilised for rotating a directional vector.

```
float4 GetQuaternionRotationFromAxisAndAngle(float3 axisXYZ, float angleDegrees)
{
    float4 qr;
    float half_angle = (angleDegrees * 0.5) * 3.14159 / 180.0;
    qr.x = axisXYZ.x * sin(half_angle);
    qr.y = axisXYZ.y * sin(half_angle);
    qr.z = axisXYZ.z * sin(half_angle);
    qr.w = cos(half_angle);
    return qr;
}

float4 QuaternionMultiplication(float4 quaternion1, float4 quaternion2)
{
    float4 qr;
    qr.x = (quaternion1.w * quaternion2.x) + (quaternion1.x * quaternion2.w) + (quaternion1.y * quaternion2.z) - (quaternion1.z * quaternion2.y);
    qr.y = (quaternion1.w * quaternion2.y) - (quaternion1.x * quaternion2.z) + (quaternion1.y * quaternion2.w) + (quaternion1.z * quaternion2.x);
    qr.z = (quaternion1.w * quaternion2.z) + (quaternion1.x * quaternion2.y) - (quaternion1.y * quaternion2.x) + (quaternion1.z * quaternion2.w);
    qr.w = (quaternion1.w * quaternion2.w) - (quaternion1.x * quaternion2.x) - (quaternion1.y * quaternion2.y) - (quaternion1.z * quaternion2.z);
    return qr;
}

float4 GetQuaternionConjugate(float4 quaternion)
{
    return float4(-quaternion.x, -quaternion.y, -quaternion.z, quaternion.w);
}

float3 RotateDirectionAroundOrigin(float3 directionVector, float3 axis, float angle)
{
    float4 q = GetQuaternionRotationFromAxisAndAngle(axis, angle);
    float3 v = directionVector.xyz;
    return v + 2.0 * cross(q.xyz, cross(q.xyz, v) + q.w * v);
}
```

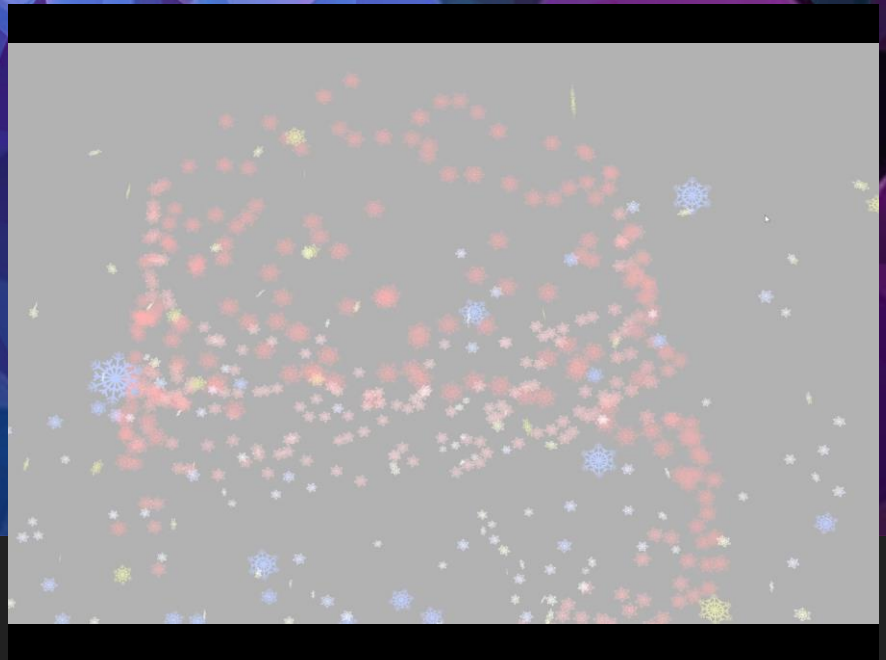
Quaternions in the Compute Shader

- Quaternion rotations allow for a rotational velocity to be applied to a particle's travel direction, allowing for curved moment of a particle.
- In the compute shader, this function is called to process the aforementioned velocity for all particles. Creating visuals such as a red spiral, using the same functionality for rotating particle quads:

```
// Rotations and Direction //

#ifdef DEFINE_USECONDENSEDPARTICLEDATA
float3 RotateDirection(min16float3 dDir)
#else
float3 RotateDirection(float3 dDir)
#endif
{
    if (dDirRotVel.x == dDirRotVel.y == dDirRotVel.z == 0.0)
    {
        return dDir;
    }

    float3 qI = normalize(dDirRotVel);
    float qR = length(dDirRotVel) * gDeltaTime;
    dDir = RotateDirectionAroundOrigin(dDir, qI, qR);
    return dDir;
}
```



Potential Improvements & Alternative Approaches

Key Optimisations:

- Decreasing the size of the ParticleData struct to increase speed.
  - DirectX and HLSL possess a 32-bit data type named “Half”, which can represent floating point numbers.
  - Testing reveals this reduced the size of ParticleData from 72 bytes to 48 bytes.
    - This means ParticleData wouldn’t cross 64-byte cache lines anymore.
  - Size can be reduced even further with smaller data types, such as a “Nibble” (0.5 bytes).
- Constructing the Vertex Data on the GPU.
  - This is a critical bottleneck on the CPU, constructing these on the GPU is a better alternative.
- There is potential in processing tasks (where relevant) on the Geometry Shader as an alternative to the Compute Shader.

Future Functionality/Features:

- Texture Diversity - Supporting different textures for each type of particle, such leaves and flames.
- Blending Improvements - Allows for more convincing and flexible transparency, enabling more visual effects.
- Particles Shadows - Essential for realistic particle effects.

Thank You