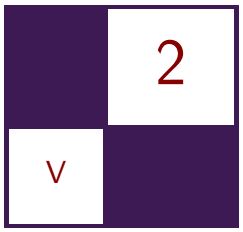

Implementing a GPU-Only Particle-Collision System with ASTC 3D Textures and OpenGL ES 3.0

Daniele Di Donato

2.1 Introduction

Particle simulation has always been a part of games to realize effects that are difficult to achieve in a rasterizer systems. As the name suggests, particles are associated with the concept of small elements that appear in huge numbers. To avoid the complexity of real-world physics, the particles used in graphics tend to be simplified so they can be easily used in real-time applications. One of these simplifications is to consider each particle independent and not interacting with each other, which makes them suitable for parallelization across multiple processors.

The latest mobile GPUs support OpenGL ES 3.0, and the new features added gives us the right tools for implementing this simulation. We also wanted to enable a more realistic behavior, especially concerning collisions with objects in the scene. This can be computationally expensive and memory intensive since the information of the geometry needs to be passed to the GPU and traversed, per simulation step, if we want to parallelize the traditional CPU approach. With the introduction of ASTC [Nystad et al. 12] and its support for 3D textures, we are now able to store voxelized data on mobile devices with huge memory savings. This texture can be used in the OpenGL pipeline to read information about the scene and use it to modify the particle's trajectory at the cost of a single texture access per particle. The following sections describe all the steps of the particle-system simulation in detail (Figure 2.1).

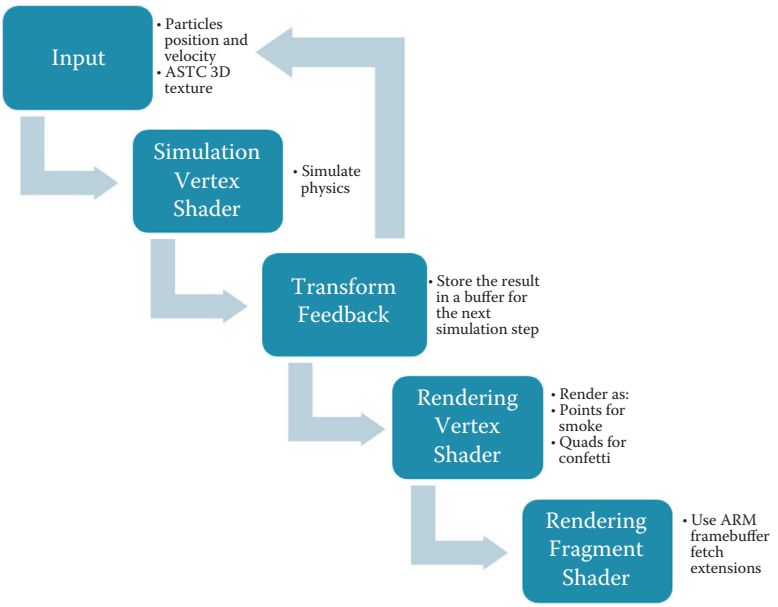

Figure 2.1. Simulation steps.

2.2 GPU-Only Particle System

2.2.1 Gathering Information about the Surroundings

To handle collisions with objects, we need to give the particles knowledge of their surroundings. This is achieved using a 3D texture describing uniform voxels of our 3D scene. For each voxel, we check if it's occupied by parts of the object and we store informations for that location. For voxels that end up on the surface of the mesh, we store a normal direction, while for internal voxels, we store the direction to the nearest surface and the amount of displacement from the current voxel to the nearest voxel on the surface. To achieve this, we used a freely available software called Voxelizer [Morris 13]. Voxelizer uses 32-bit floats for the mentioned values, so we convert them to 16-bit half-floats. This reduces the space needed by the data to be stored in a 3D texture. ASTC allows converting 16-bit per channel values, representing half-floats in our case, for the same memory cost. This gives us a better precision compared to using 8-bit values.

2.2.2 Compression Using ASTC 3D

The ASTC texture compression format is a block-based compression algorithm that is able to compress 2D and 3D textures in LDR or HDR format. Compared to

Block Dimension	Bit Rate (bits per pixel)
$3 \times 3 \times 3$	4.74
$4 \times 3 \times 3$	3.56
$4 \times 4 \times 3$	2.67
$4 \times 4 \times 4$	2.0
$5 \times 4 \times 4$	1.60
$5 \times 5 \times 4$	1.28
$5 \times 5 \times 5$	1.02
$6 \times 5 \times 5$	0.85
$6 \times 6 \times 5$	0.71
$6 \times 6 \times 6$	0.59

Table 2.1. ASTC 3D available block sizes.

other compression algorithms, ASTC offers more parameters to tune the quality of the final image (more details are available in [Smith 14]). The main options are the block size, the quality settings, and an indication of the correlation within the color channels (and the alpha channel if present). For the 3D format, ASTC allows the block sizes described in Table 2.1

Because the block compressed size is always 128 bits for all block dimensions and input formats, the bit rate is simply 128/(number of texels in a block). This specifies the tradeoff between quality and dimension of the generated compressed texture. In Figure 2.2, various ASTC compressed 3D texture have been rendered using slicing planes and various block sizes.

The other parameter to choose is the amount of time spent finding a good match for the current block. From a high-level view, this option is used to increase the quality of the compression at the cost of more compression time. Because this is typically done as an offline process, we can use the fastest option for debug

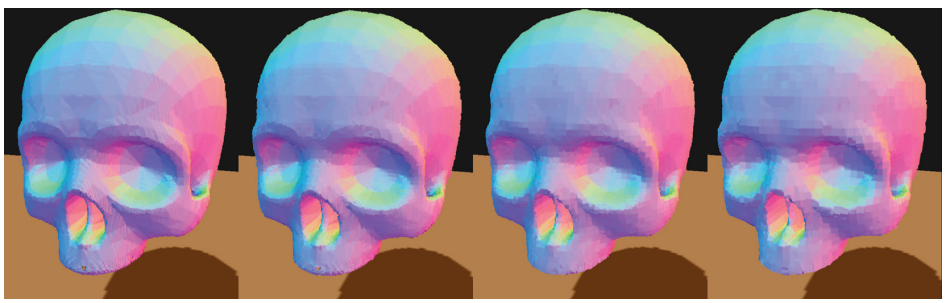

Figure 2.2. From left to right: uncompressed 3D texture, ASTC 3D $3 \times 3 \times 3$ compressed texture, ASTC 3D $4 \times 4 \times 4$ compressed texture, and ASTC 3D $5 \times 5 \times 5$ compressed texture.

purposes and compress using the best one for release. The options supported by the free ARM ASTC evaluation codec [ARM Mali 15a, ARM Mali 15c] are *very fast*, *fast*, *standard*, *thorough*, and *exhaustive*. The last parameter to set is the correlation within the color channels. The freely available tools also allows us to use various preset configuration options based on the data you want to compress. For example, the tool has a preset for 2D normal maps compression that treats the channels as uncorrelated and also uses a different error metric for the conversion. This preset is not available for 3D textures, so we set the uncorrelation using the fine-grained options available. Note that the ASTC compression tool used does not store negative numbers, even in case of half-float format. This is due to the internal implementation of the ASTC algorithm. Because our data contains mostly unit vectors, we shifted the origin to be at $[1, 1, 1]$ so that the vectors resides in the $[0, 0, 0]$ to $[2, 2, 2]$ 3D cube.

2.2.3 Statistics of the Savings

Compressing the 3D texture using ASTC gave us a huge amount of memory saving, especially thanks to its ability to compress HDR values at the same cost as LDR values. As can be seen from Table 2.2, the memory saving can reach nearly 90% with the subsequent reduction of memory read bandwidth, and hence energy consumption. The memory read bandwidth has been measured using ARM Streamline profiling tool on a Samsung Galaxy Note 10.1, 2014 edition. We measured the average read bandwidth from the main memory to the L2 cache of the GPU running the demo for around two minutes for each ASTC texture format we used. The energy consumption per frame is an approximation computed using ARM internal reference values for DDR2 and DDR3 memory modules.

2.3 Physics Simulation

The physics simulation is really simple and tries to approximate the physical behavior. Each particle will be subject to the force of gravity as well as other forces we choose to apply. Given an initial state $t = 0$ for the particles, we simulate the second law of motion and compute the incremental movement after a Δt. The Δt used in the demo is fixed to 16 ms since we assume the demo will run at 60 fps. Methods that try to solve ordinary and partial derivative equations using incremental steps are typically called *explicit methods*.

2.3.1 Explicit Methods

To delegate the physics computation to the GPU, we decided to use an explicit method of computation for the simulation step, since this methods fits well with

	Sphere	**Skull**	**Chalice**	**Rock**	**Hand**
Texture Resolution	$128 \times 128 \times 128$	$180 \times 255 \times 255$	$255 \times 181 \times 243$	$78 \times 75 \times 127$	$43 \times 97 \times 127$
Texture Size MB					
Uncompressed	16.78	82.62	89.73	5.94	4.24
ASTC $3 \times 3 \times 3$	1.27	6.12	6.72	0.45	0.34
ASCT $4 \times 4 \times 4$	0.52	2.63	2.87	0.19	0.14
ASTC $5 \times 5 \times 5$	0.28	1.32	1.48	0.10	0.07
Memory Read Bandwidth in MB/s					
Uncompressed	644.47	752.18	721.96	511.48	299.36
ASTC $3 \times 3 \times 3$	342.01	285.78	206.39	374.19	228.05
ASCT $4 \times 4 \times 4$	327.63	179.43	175.21	368.13	224.26
ASTC $5 \times 5 \times 5$	323.10	167.90	162.89	366.18	222.76
Energy consumption per frame DDR2 mJ per frame					
Uncompressed	4.35	5.08	4.87	3.45	2.01
ASTC $3 \times 3 \times 3$	2.31	1.93	1.39	2.53	1.54
ASCT $4 \times 4 \times 4$	2.21	1.21	1.18	2.48	1.51
ASTC $5 \times 5 \times 5$	2.18	1.13	1.10	2.47	1.50
Energy consumption per frame DDR3 mJ per frame					
Uncompressed	3.58	4.17	4.01	2.84	1.66
ASTC $3 \times 3 \times 3$	1.90	1.59	1.15	2.08	1.27
ASCT $4 \times 4 \times 4$	1.82	1.00	0.97	2.04	1.24
ASTC $5 \times 5 \times 5$	1.79	0.93	0.90	2.03	1.24

Table 2.2. ASTC 3D texture compression examples with various block sizes.

the transform feedback feature available through OpenGL ES 3.0. For the purpose of the demo, we implemented a simple Euler integration, and each shader execution computes a step of the integration. This implementation is good enough for the demo, but for advanced purposes, a variable time step can be used and each shader execution can split this time step further and compute a smaller integration inside the shader itself.

So, the physical simulation for step $N + 1$ will be dependent on a function of step N and the delta time (Δt) that occurred between the simulation steps:

$$Y(t + \Delta t) = F(Y(t), \Delta t).$$

Due to the time dependency of position, velocity, and acceleration, this method is suitable for use in our simulation.

```
typedef struct _XFormFeedbackParticle
{
    Vec3  Position;
    Vec3  Velocity;
    Vec4  Attrib;
    float Life;
} XFormFeedbackParticle;

glGenBuffers( 2, m_XformFeedbackBuffers );

glBindBuffer( GL_ARRAY_BUFFER,
              m_XformFeedbackBuffers[0] );

glBufferData( GL_ARRAY_BUFFER,
              sizeof( XFormFeedbackParticle ) *
              totalNumberOfParticles ,
              NULL,
              GL_STREAM_DRAW );

glBindBuffer( GL_ARRAY_BUFFER,
              m_XformFeedbackBuffers[1] );

glBufferData( GL_ARRAY_BUFFER,
              sizeof( XFormFeedbackParticle ) *
              totalNumberOfParticles ,
              NULL,
              GL_STREAM_DRAW );

//Initialize the first buffer with the particles'
//data from the emitters
unsigned int offset = 0;
for( unsigned int i = 0; i < m_Emitters.Length(); i++ )
{
    glBindBuffer( GL_ARRAY_BUFFER,
                  m_XformFeedbackBuffers[0] );

    glBufferSubData( GL_ARRAY_BUFFER,
                     offset,
                     m_Emitters[i]->MaxParticles() *
                     sizeof( XFormFeedbackParticle ),
                     m_Emitters[i]->Particles() );

    offset += m_Emitters[i]->MaxParticles() *
              sizeof( XFormFeedbackParticle );
}
```

Listing 2.1. Transform feedback buffers initialization.

```
const char* xformFeedbackVaryings[4] = { "oParticlePos",
                                         "oParticleVel",
                                         "oParticleAttrib",
                                         "oParticleLife"   };
glTransformFeedbackVaryings( m_XFormFeedbackShader,
                             4,
                             xformFeedbackVaryings,
                             GL_INTERLEAVED_ATTRIBS );
```

Listing 2.2. Transform feedback output varyings definition.

Position1	Velocity1	Attrib1	Life1	Position2	Velocity2	Attrib2	Life2	...

Table 2.3. Order of `GL_INTERLEAVED_ATTRIBS` attributes.

Position1	Position2	Position3	...
Velocity1	Velocity2	Velocity3	...
Attrib1	Attrib2	Attrib3	...
Life1	Life2	Life3	...

Table 2.4. Order of `GL_SEPARATE_ATTRIBS` attributes.

2.3.2 OpenGL ES Transform Feedback Overview

Transform feedback allows users to store the result of a vertex shader execution into a predefined vertex buffer. This feature fits well with the explicit methods described above, since we can simulate the various steps using two buffers that are swapped at each simulation step (this is usually called ping-ponging). After we generate IDs for the transform feedback buffers using `glGenBuffers`, we initialize them with a set of random particles. If multiple emitters are present, we can store all their particles in the same buffer so that one step of the simulation can actually update multiple emitters in the scene (see Listing 2.1).

Vertex shaders output various results to the subsequent fragment shader, so we need a way to specify which results should also be written to the predefined output buffer. This can be done after we attach the vertex program that will run the simulation to the main program.

The command `glTransformFeedbackVaryings` (see Listing 2.2) will check if the specified strings are defined as output of the vertex shaders, and `GL_INTERLEAVED_ATTRIBS` will tell OpenGL in which layout to store the data. Possible options are `GL_INTERLEAVED_ATTRIBS` and `GL_SEPARATE_ATTRIBS`. The former will store the result of the vertex shader in a single buffer and in order as specified by the strings passed to the function and the particles' data will look like Table 2.3. The latter stores each attribute in a separate buffer (see Table 2.4).

During the rendering, we do the following:

1. Set which buffer to use as the destination buffer for transform feedback using the specific `GL_TRANSFORM_FEEDBACK_BUFFER` flag.

```
glBindBufferBase( GL_TRANSFORM_FEEDBACK_BUFFER,
                  0,
                  m_XformFeedbackBuffers[1] );
```

2. Set which buffer is the source buffer and how the data is stored in it.

```
glBindBuffer( GL_ARRAY_BUFFER, m_XformFeedbackBuffers[0] );
glEnableVertexAttribArray( m_ParticlePositionLocation );
```

```
glEnableVertexAttribArray( m_ParticleVelocityLocation );
glEnableVertexAttribArray( m_ParticleAttribLocation );
glEnableVertexAttribArray( m_ParticleLifeLocation );

//We store in one buffer the 4 fields that represent a ↩
    particle
// Position: 3 float values for a total of 12 bytes
// Velocity: 3 float values for a total of 12 bytes
// Attrib: 2 float values for a total of 8 bytes
// life: 1 float value for a total of 4 bytes

glVertexAttribPointer( m_ParticlePositionLocation,
                       3,
                       GL_FLOAT,
                       GL_FALSE,
                       sizeof( XFormFeedbackParticle ),
                       0);

glVertexAttribPointer( m_ParticleVelocityLocation,
                       3,
                       GL_FLOAT,
                       GL_FALSE,
                       sizeof( XFormFeedbackParticle ),
                       12);

glVertexAttribPointer( m_ParticleAttribLocation,
                       4,
                       GL_FLOAT,
                       GL_FALSE,
                       sizeof( XFormFeedbackParticle ),
                       24);

glVertexAttribPointer( m_ParticleLifeLocation,
                       1,
                       GL_FLOAT,
                       GL_FALSE,
                       sizeof( XFormFeedbackParticle ),
                       40);
```

3. Enable transform feedback and disable the rasterizer step. The former is done using the `glBeginTransformFeedback` function to inform the OpenGL pipeline that we are interested in saving the results of the vertex shader execution. The latter is achieved using the `GL_RASTERIZER_DISCARD` flag specifically added for the transform feedback feature. This flag disables the generation of fragment jobs so that only the vertex shader is executed. We disabled the fragment execution since the rendering of the particles required two different approaches based on the scene rendered and splitting the simulation from the rendering gave us a cleaner code base to work with.

```
glEnable( GL_RASTERIZER_DISCARD );
glBeginTransformFeedback( GL_POINTS );
```

4. Render the particles as points.

```
glDrawArrays( GL_POINTS, 0, MaxParticles );
```

5. Disable transform feedback and re-enable the rasterizer.

```
glEndTransformFeedback();
glDisable( GL_RASTERIZER_DISCARD );
```

2.3.3 Manage the Physics in the Vertex Shader Using 3D Textures

The attributes of each particle are read in the vertex shader as vertex attributes and used to compute the next incremental step in the physics simulation. First, we compute the total forces acting on the particles. Since this is a very simple simulation, we ended up simulating just the gravity, a constant force, and the air friction. The air friction is computed using the Stokes' drag formula [Wikipedia 15] because the particles are considered to be small spheres:

$$Fd = -6\pi\eta rv,$$

where η is the dynamic viscosity coefficient of the air and is equal to 18.27 μ Pa, r is the radius of the particle (we used 5 μm in our simulation), and v is the velocity of the particle. Since the first part of the product remains constant, we computed it in advance to avoid computing it per particle.

```
//Air friction is given by 6.0 * 3.14 * 5 * 0.000018 = 0.0016956
vec3 totalForce = uConstantForce +
                  ( uParticleMass * gravity ) -
                  0.0016956 * iParticleVel;

vec3 totalAcceleration = totalForce/uParticleMass;

oparticlePos_worldSpace = iparticle_Pos +
                          ( iparticle_Vel * uDeltaT ) +
                          ( totalAcceleration * uDeltaTSquared );
```

The new position is then transformed using the transformation matrix derived by the bounding box of the model. This matrix is computed to have the bounding box minimum to be the origin $(0, 0, 0)$ of the reference. Also, we want the area of world space inside the bounding box to be mapped to the unit cube space $(0, 0, 0)$–$(1, 1, 1)$. Applying this matrix to the particle's position in world space gives us the particle's coordinate in the space with the origin at the minimum corner of the bounding box and also scaled based on the dimension of the model. This means that the particles positioned in bounding box space within $(0, 0, 0)$

and $(1, 1, 1)$ have a chance to collide with the object, and this position is actually the 3D texture coordinate we will use to sample the 3D texture of the model.

- Host code.

```
uBoundingBoxMatrix = ( (1.0/max.x-min.x, 0.0,0.0,-min.x),
                       (0.0, 1.0/max.y-min.y, 0.0, -min.y),
                       (0.0, 0.0, 1.0/max.z-min.z, -min.z),
                       (0.0, 0.0, 0.0, 1.0) ) *
                     inverse( ModelMatrix );
```

- Vertex shader.

```
vec4 oParticlePos_BBSpace = uBoundingBoxMatrix *
                            vec4( oparticlePos_worldSpace, 1.0 );

vec4 surfaceNormal =  texture( uCollisionTexture, tex3dCoord );
```

The surface's normal will be encoded in a 32-bit field and stored to be used later in the rendering pass to orient the particles in case of collisions. Due to the discrete nature of the simulation, it can happen that a particle goes inside the object. We recognize this event when sampling the 3D texture since we store a flag plus other data in the alpha channel of the 3D texture. When this event happens, we use the gradient direction stored in the 3D texture plus the amount of displacement that needs to be applied and we "push" the particle to the nearest surface. The push is applied to the particles in the bounding-box space, and the inverse of the `uBoundingBoxMatrix` is then used to move the particles back to the world space. Discrete time steps can cause issues when colliding with completely planar surfaces since a sort of swinging can appear, but at interactive speeds ($\geq$ 30 fps), this is almost unnoticeable. For particles colliding with the surface of the object, we compute the new velocity direction and magnitude using the previous velocity magnitude, the surface normal, the surface tangent direction, and a bouncing resistance to simulate different materials and particle behavior. We use the particle's mass as sliding factor so that heavier particles will bounce while lighter particles such as dust and smoke will slide along the surface. A check needs to be performed for the tangent direction since the normal and the velocity can be parallel, and in that case, the cross product will give an incorrect result (see Listing 2.3).

The velocity is then used to move the particle to its new position. Because we want to avoid copying memory within the GPU and CPU, the lifetime of all the particles should be managed in the shader itself. This means we check if the lifetime reached 0 and reinitialize the particle attributes such as initial position, initial velocity, and total particle duration. To make the simulation more

```
float slidingFactor = clamp( uParticleMass, 0.0,1.0 );
vec3 velocityDir =  normalize( iparticle_Vel );
vec3 tangentDir = cross( surfaceNormal.xyz, velocityDir );

if( length(tangentDir) < 0.0001 )
{
    tangentDir = getRandomTangentDir( surfaceNormal.xyz, 0.0 );
}

iparticle_Vel = length( iparticle_Vel ) *
                ( surfaceNormal.xyz * slidingFactor +
                  tangentDir.xyz * ( 1.0-slidingFactor ) ) *
                uBouncingResistance;
```

Listing 2.3. Particle-collision behavior.

interesting, some randomness can be added while the particles are flowing and no collision occurred. The fragment shader of the simulation is actually empty. This is understandable since we do not need to execute any fragment work for the simulation results. Also, we have enabled the `GL_RASTERIZER_DISCARD` to skip all fragment work from being executed. In a way that differs from the OpenGL standard, OpenGL ES needs a fragment shader to be attached to the program, even if is not going to be used.

2.4 Rendering the Particles

After updating the particles' locations, we can render them as we want. In our demo, we decided to render them as smoke particles and as confetti. The light lamp shape on the floor is procedurally generated using its texture coordinates. The shadows are created using a projected texture that is generated from the light point of view. This texture is used for the shadows of the floor as well the ones on the objects. To achieve this we implement an incremental approach:

1. Render the object without color enabled so that its depth is stored in the depth buffer. We need to do this step to prevent particles behind the object (from the point of view of the light) from casting shadows on the object.

2. Render the particles with depth testing on, but not depth writing.

3. Render the object normally using the texture generated at Step 2 for the shadows.

4. Render the object as shadow in the texture from Step 2.

5. Render the floor with the result of Step 4 for the shadows.

This approach can be optimized. For example, we can use two different framebuffers for the shadow of the floor and on the object so that we avoid incremental renderings (refer to [Harris 14] for more information). To achieve this, we copy the result of the texture created at the end of Step 2 into the other framebuffer and then render the object as shadow on it.

2.4.1 Smoke Scene

In this scene, the smoke (Figure 2.3) is rendered as point sprites since we always want them to face the viewpoint. The smoke is rendered using a noise texture and some mathematics to compute the final color as if it was a 3D volume. To give the smoke a transparent look, we need to combine different overlapping particles' colors. To do so, we used blending and disabled the Z-test when rendering the particles. This gives a nice result, even without sorting the particles based on the Z-value (otherwise we have to map the buffer in the CPU). Another reason for disabling it is to achieve soft particles. From Mali-T600 GPUs onward, we can use a specific extension in the fragment shader called `GL_ARM_shader_framebuffer_fetch` to read back the values of the framebuffer (color, depth, and stencil) without having to render to a texture [Björge 14]. The extension allows us to access a set of built-in variables (`gl_LastFragColorARM`, `gl_LastFragDepthARM`, `gl_LastFragStencilARM`) from the fragment shader, and for each pixel, the value is based on previous rendering results.

```
#extension GL_ARM_shader_framebuffer_fetch_depth_stencil : enable
#ifdef GL_ARM_shader_framebuffer_fetch_depth_stencil
float dla= (2.0 * uNear) /
           (uFar + uNear - gl_LastFragDepthARM * (uFar - uNear));
#else
   //Texture read fallback
#endif
```

This feature makes it easier to achieve soft particles, and in the demo, we use a simple approach. First, we render all the solid objects so that the Z-value will be written in the depth buffer. Afterward, we render the smoke and we can read the depth value of the object and compare it with the current fragment of the particle (to see if it is behind the object) and fade the color accordingly. This technique eliminates the sharp profile that is formed by the particle quad intersecting the geometry due to the Z-test. During development, the smoke effect looked nice, but we wanted it to be more dense and blurry. To achieve all this, we decided to render the smoke in an offscreen render buffer with a lower resolution compared to the main screen. This gives us the ability to have a blurred smoke (since the lower resolution removes the higher frequencies) as well as lets us increase the number of particles to get a denser look. The current implementation uses a 640×360 offscreen buffer that is up-scaled to 1080p resolution in the final image.

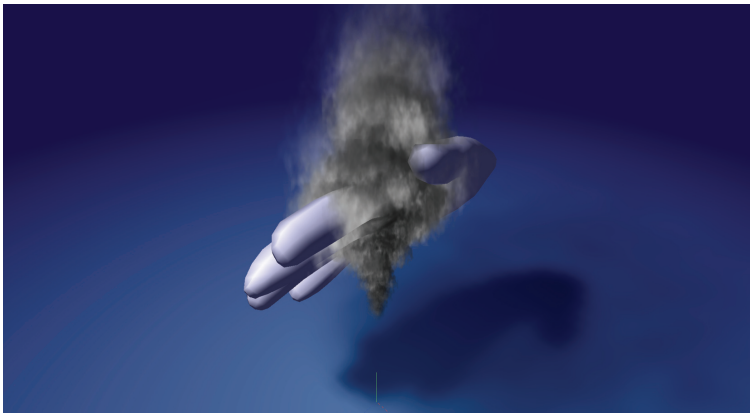

Figure 2.3. Smoke scene.

A naïve approach causes jaggedness on the outline of the object when the smoke is flowing near it due to the blending of the up-sampled low-resolution buffer. To minimize this effect, we apply a bilateral filter. The bilateral filter is applied to the offscreen buffer and is given by the product of a Gaussian filter in the color texture and a linear weighting factor given by the difference in depth. The depth factor is useful on the edge of the model because it gives a higher weight to neighbor texels with depth similar to the one of the current pixel and lower weight when this difference is higher. (If we consider a pixel on the edge of a model, some of the neighbor pixels will still be on the model while others will be far in the background.)

2.4.2 Confetti Scene

In this case, we used quads instead of points since we needed to rotate the particles when they slide along the surfaces (Figure 2.4). Those quads are initialized to $\texttt{min} = (-1, -1, 0)$ and $\texttt{max} = (1, 1, 0)$. The various shapes are achieved procedurally checking the texture coordinates of the quad pixels. To rotate the quad accordingly, we retrieve the normal of the last surface touched and compute the tangent and binormal vectors. This gives us a matrix that we use to rotate the initial quad position, and afterward we translate this quad into the position of the particle that we computed in the simulation step.

2.4.3 Performance Optimization with Instancing

Even if the quad data is really small, they waste memory because the quads are all initialized with the same values and they all share the same number of vertices and texture coordinates. The instancing feature introduced in OpenGL ES 3.0

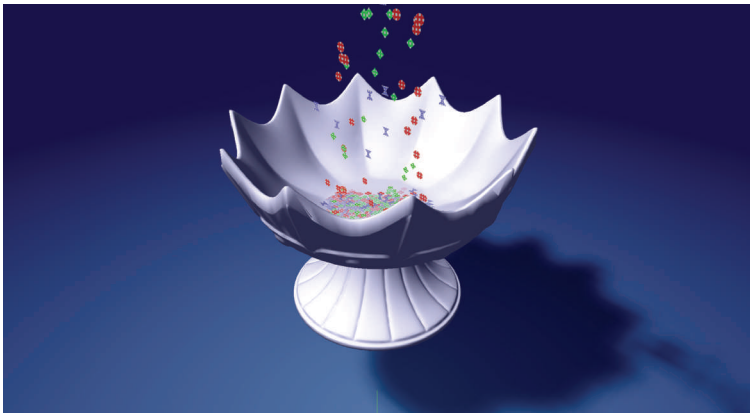

Figure 2.4. Confetti scene.

allows us to avoid replication of vertex attribute by defining just one "template" of the mesh we want to render. This template is then instantiated multiple times and the user will vary the parameters (matrices, colors, textures, etc.) to represent multiple meshes with different characteristic with a single draw call (Figure 2.5).

OpenGL ES instancing overview.

1. Bind the buffers that we will use as the template source data.

```
glBindBuffer( GL_ARRAY_BUFFER, m_QuadPositionBuffer );
glEnableVertexAttribArray( m_QuadPositionLocation );
glVertexAttribPointer( m_QuadPositionLocation,
                       3,
                       GL_FLOAT,
                       GL_FALSE,
                       0,
                       (void*)0 );

//Set up quad texture coordinate buffer
glBindBuffer( GL_ARRAY_BUFFER, m_TexCoordBuffer );
glEnableVertexAttribArray( m_QuadTexCoordLocation );
glVertexAttribPointer( m_QuadTexCoordLocation,
                       2,
                       GL_FLOAT,
                       GL_FALSE,
                       0,
                       (void*)0 );
```

2. Set a *divisor* for each vertex attribute array. The divisor specifies how the vertex attributes advance in the array when rendering instances of primitives in a single draw call. Setting it to 0 will make the attribute advance

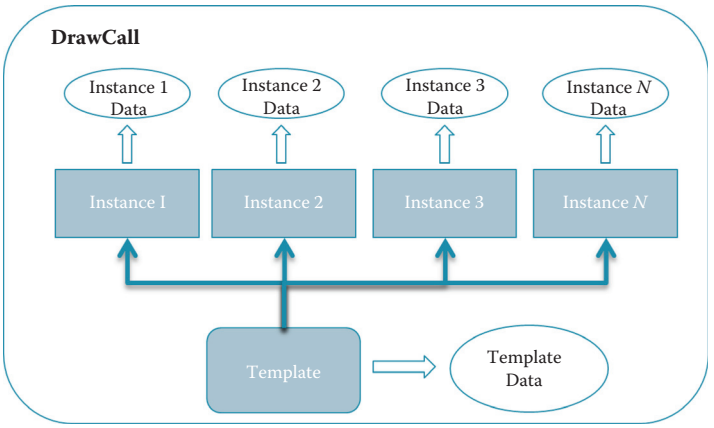

Figure 2.5. OpenGL ES 3.0 instancing.

once per vertex, restarting at the start of each instance rendered. This is what we want to happen for the initial quad position and texture coordinate since they will be the same for each particle (instance) rendered.

```
glVertexAttribDivisor( m_QuadPositionLocation, 0 );
glVertexAttribDivisor( m_QuadTexCoordLocation, 0 );
```

3. For the attributes computed in the simulation step, we would like to shift the vertex buffer index for each of the particles (instances) to be rendered. This is achieved using a divisor other than zero. The divisor then specifies how many instances should be rendered before we advance the index in the arrays. In our case, we wanted to shift the attributes after each instance is rendered, so we used a divisor of 1.

```
glVertexAttribDivisor( m_UpdatedParticlePosLocation, 1 );
glVertexAttribDivisor( m_UpdatedParticleLifeLocation, 1 );
glVertexAttribDivisor( m_UpdatedParticleAttribLocation, 1 );
```

4. Bind the buffer that was output from the simulation step. Set up the vertex attributes to read from this buffer.

```
glBindBuffer( GL_ARRAY_BUFFER, m_XformFeedbackBuffers[1] );

glVertexAttribPointer( m_UpdatedParticlePosLocation,
                       3,
                       GL_FLOAT,
```

```
                        GL_FALSE,
                        sizeof( XFormFeedbackParticle ),
                        0);

glVertexAttribPointer( m_UpdatedParticleAttribLocation,
                        4,
                        GL_FLOAT,
                        GL_FALSE,
                        sizeof( XFormFeedbackParticle ),
                        24);

glVertexAttribPointer( m_UpdatedParticleLifeLocation,
                        1,
                        GL_FLOAT,
                        GL_FALSE,
                        sizeof( XFormFeedbackParticle ),
                        40);
```

5. Render the particles (instances). The function allows to specify how many vertices belong to each instance and how many instances we want to render. Note that when using instancing, we are able to access a built-in variable `gl_InstanceID` inside the vertex shader. This variable specifies the ID of the instance we are currently rendering and can be used to access uniform buffers.

```
glDrawArraysInstanced( GL_TRIANGLE_STRIP, 0, 4, MaxParticles );
```

6. Always set back to 0 the divisor for all the vertex attribute arrays since they can affect subsequent rendering even if we are not using indexing.

```
glDisableVertexAttribArray( m_QuadPositionLocation );
glDisableVertexAttribArray( m_QuadTexCoordLocation );
glVertexAttribDivisor( m_UpdatedParticlePosLocation, 0 );
glVertexAttribDivisor( m_UpdatedParticleAttribLocation, 0 );
glVertexAttribDivisor( m_UpdatedParticleLifeLocation, 0 );
```

2.5 Conclusion

Combining OpenGL ES 3.0 features enabled us to realize a GPU-only particle system that is capable of running at interactive speeds on current mobile devices. The techniques proposed are experimental and have some drawbacks, but the reader can take inspiration from this chapter and explore other options using ASTC LDR/HDR/3D texture as well as OpenGL ES 3.0. In case there is need to sort the particles, the compute shader feature recently announced in the OpenGL ES 3.1 specification will enable sorting directly on the GPU.

An issue derived from the use of a texture is the texture's resolution. This technique can describe a whole 3D static scene in a single 3D texture, but the resolution of it needs to be chosen carefully since too small resolution can cause parts of objects to not collide properly since multiple parts with different normals will be stored in the same voxel. Also, space is wasted if the voxelized 3D scene contains parts with no actual geometry in them but that fall inside the volume that is voxelized. Since we are simulating using a discrete time step, issues can appear if we change the system too fast. For example, we can miss the collision detection in narrow parts of the object if we rotate it too fast.

Bibliography

[ARM Mali 15a] ARM Mali. "ASTC Evaluation Codec." http://malideveloper.arm.com/develop-for-mali/tools/astc-evaluation-codec, 2015.

[ARM Mali 15b] ARM Mali. "Mali Developer Center." http://malideveloper.arm.com, 2015.

[ARM Mali 15c] ARM Mali. "Mali GPU Texture Compression Tool." http://malideveloper.arm.com/develop-for-mali/tools/asset-creation/mali-gpu-texture-compression-tool/, 2015.

[Björge 14] Marius Björge. "Bandwidth Efficient Graphics with ARM Mali GPUs." In *GPU Pro 5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 275–288. Boca Raton, FL: CRC Press, 2014.

[Harris 14] Peter Harris. "Mali Performance 2: How to Correctly Handle Framebuffers." *ARM Connected Community*, http://community.arm.com/groups/arm-mali-graphics/blog/2014/04/28/mali-graphics-performance-2-how-to-correctly-handle-framebuffers, 2014.

[Morris 13] Dan Morris. "Voxelizer: Floodfilling and Distance Map Generation for 3D Surfaces." http://techhouse.brown.edu/~dmorris/voxelizer/, 2013.

[Nystad et al. 12] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. "Adaptive Scalable Texture Compression." In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, pp. 105–114. Aire-la-ville, Switzerland: Eurographics Association, 2012.

[Smith 14] Stacy Smith. "Adaptive Scalable Texture Compression." In *GPU Pro 5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 313–326. Boca Raton, FL: CRC Press, 2014.

[Wikipedia 15] Wikipedia. "Stokes' Law." http://en.wikipedia.org/wiki/Stokes%27_law, 2015.