Real-time persistent mesh painting with GPU particle systems

Andreas Larsson
Master of Science Thesis in Computer Engineering

Real-time persistent mesh painting with GPU particle systems
Andreas Larsson
LiTH-ISY-EX–17/5027–SE

Supervisor: Axel Kinner
Fatshark AB

Examiner: Ingemar Ragnemalm
ISY, Linköpings universitet

Division of Information Coding
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2016 Andreas Larsson
Abstract

Particle systems are used to create visual effects in real-time applications such as computer games. However, emitted particles are often transient and do not leave a lasting impact on a 3D scene. This thesis work presents a real-time method that enables GPU particle systems to paint meshes in a 3D scene as the result of particle collisions, thus adding detail to and leaving a lasting impact on a scene. The method uses screen space collision detection and a mapping from screen space to texture space of meshes to determine where to apply paint. The method was tested for its time complexity and how well it performed in scenarios similar to those found in computer games. The results show that the method probably can be used in computer games. Performance and visual fidelity of the paint application is not directly dependent on the amount of simulated particles, but depends only on the complexity of the meshes and their texture mapping as well as the resolution of the paint. It is concluded that the method is renderer agnostic and could be added to existing GPU particle systems and that other types of effects than those showed in the thesis could be achieved by using the method.
Acknowledgments

Many thanks to the people at Fatshark who treated me as one of their own, especially my supervisor Axel Kinner and my friend who introduced me to the company. I would also like to thank my friends in Stockholm who invited me to stay in their homes for the duration of my thesis work, my examiner Ingemar Ragnemalm for handling the administrative part of the thesis work, Fredrik Präntare who agreed to be my opponent and to my family who provided unconditional love and support during this time.

Linköping, Januari 2016
Andreas Larsson
## Contents

<table>
<thead>
<tr>
<th>Notation</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Aim</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research questions</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Delimitations</td>
<td>3</td>
</tr>
<tr>
<td>2 Theory</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Graphics pipeline</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Overview of Direct3D 11</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Coordinate spaces</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Deferred rendering</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Texture sampling</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Nearest neighbor</td>
<td>8</td>
</tr>
<tr>
<td>2.3.2 Linear</td>
<td>8</td>
</tr>
<tr>
<td>2.3.3 Cubic with B-spline weights</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Billboards</td>
<td>10</td>
</tr>
<tr>
<td>2.4.1 World oriented</td>
<td>10</td>
</tr>
<tr>
<td>2.4.2 View point oriented</td>
<td>10</td>
</tr>
<tr>
<td>2.4.3 View point oriented, axial</td>
<td>11</td>
</tr>
<tr>
<td>2.4.4 View plane oriented</td>
<td>11</td>
</tr>
<tr>
<td>2.4.5 View plane oriented, axial</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Particle systems</td>
<td>11</td>
</tr>
<tr>
<td>2.5.1 Stateless particles</td>
<td>12</td>
</tr>
<tr>
<td>2.5.2 Stateful particles</td>
<td>12</td>
</tr>
<tr>
<td>2.5.3 Particle representation</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Screen space collisions</td>
<td>13</td>
</tr>
<tr>
<td>2.7 Mesh material manipulation</td>
<td>14</td>
</tr>
<tr>
<td>2.8 Blending</td>
<td>15</td>
</tr>
<tr>
<td>2.9 Noise</td>
<td>16</td>
</tr>
<tr>
<td>2.10 Texture splatting</td>
<td>16</td>
</tr>
</tbody>
</table>
3 Method 19
  3.1 Rendering framework overview .................................. 19
  3.1.1 Deferred geometry pass ........................................... 20
  3.1.2 Deferred lighting pass ............................................ 20
  3.2 Particle system framework ......................................... 21
  3.3 Particle system shaders ............................................ 22
  3.3.1 Initialization step .................................................. 23
  3.3.2 Emission step .................................................... 23
  3.3.3 Updating step .................................................... 24
  3.3.4 Rendering step ................................................... 25
  3.4 Particle painting ................................................... 26
  3.4.1 Overview and setup .............................................. 26
  3.4.2 Paint shader program ............................................ 28

4 Results 31
  4.1 Painting ............................................................... 32
  4.2 GPU particle system ................................................ 32
  4.3 Real-time mesh painting with GPU particle systems ........... 33

5 Discussion 39
  5.1 Results ............................................................... 39
  5.2 Method .............................................................. 41

6 Conclusions 43

Bibliography 45
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>AABB</td>
<td>Axis-Aligned Bounding Box</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSV</td>
<td>Depth Stencil View</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>MRT</td>
<td>Multiple Render Targets</td>
</tr>
<tr>
<td>NDC</td>
<td>Normalized Device Coordinate</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td>RTV</td>
<td>Render Target View</td>
</tr>
<tr>
<td>SRV</td>
<td>Shader Resource View</td>
</tr>
<tr>
<td>UAV</td>
<td>Unordered Access View</td>
</tr>
<tr>
<td>VRAM</td>
<td>Video RAM</td>
</tr>
</tbody>
</table>
Particle systems are used in many computer applications such as computer games, motion picture and simulation of various physical phenomena. The first mention of particle system for visual effects was made by Reeves [17] in the film industry; while much of Reeves basic model for particle systems still holds, the use cases of particle systems and the technologies used to implement them have evolved. A particle system typically consist of one or more particle emitters which emit individual particles with certain behaviors and appearances, thus creating a particle effect. In computer games, particles can be used to simulate environmental effects such as smoke, fire, sparks and debris. They can also be used to visualize game elements such as energy shields, area effects etc. These effects add to the appearance of a scene by creating more interesting visuals and also notifying the user about certain events.

1.1 Motivation

As computer games are complex real-time graphics applications which have strict frame rate requirements to provide a pleasant user experience, the amount of computations which can be performed between rendered frames are limited. Early implementations of particle systems in computer games have been processed by the central processing unit (CPU) along with the other logic of the application, severely limiting the amount of particles that can be simulated due to the strict frame rate requirement. Many of today’s particle systems are processed entirely by the graphics processing unit (GPU) which provides a high amount of parallelism. Processing particles that are independent of each other is an embarrassingly parallel problem that fits the GPU architecture well. Particle systems have been implemented on the GPU for some years now. By encoding particle data into textures, a large number of particles can be simulated without needing to
leave video RAM (VRAM) between computations. [9] Recent GPU technologies such as compute shaders and read-write buffers enables parallel computations on buffers containing arbitrary data, allowing more flexibility than encoding data into textures. With GPUs becoming more powerful, newer techniques lead to increased computing power which can be used to simulate particle systems with millions of particles that interacts with a scene in many ways. As the particle size decreases and the number of particles increases it is possible to improve the visuals and create new types of effects. Particle systems on the GPU have been implemented successfully many times, what sets them apart are their features.

Particles in computer games are often transient in the sense that they do not leave a lasting impact on a scene. If interactions between meshes and particles in a scene had the ability to change the material properties of the meshes, or painting them, the particles would be able to leave a lasting impact on a scene. For example, a fire effect could scorch the surface it touches and a rain effect may create puddles over time.

1.2 Aim

This thesis aims to find a method for particles emitted from a GPU driven particle system to perform mesh material manipulation in a 3D scene, essentially painting meshes as a result of a particle collision. It shall be possible to paint meshes with a large quantity of particles and the applied paint should be persistent between frames. This effect would be able to dynamically add fine details to a 3D scene. Because of the frame rate constraint of computer games, the method shall be able to run at interactive frame rates on current GPU hardware. In this thesis, interactive frame rates are defined as more than 60 frames per second, which is often the lower bound for computer games on PC. More specifically, the method shall work on GPU hardware supporting DirectX 11 in a deferred renderer, which is used by Fatshark Games where this thesis work is performed.

1.3 Research questions

To reach the goal this thesis aims for, the following research questions need to be answered:

- **Is there a method that supports manipulation of materials, or painting, meshes in a 3D scene with thousands of independent particles?**
  In computer games, 3D scenes often have hundreds of meshes as well.

- **What is the time complexity of the found method?**
  The method on its own need to adhere to the frame rate constraints, in other words it needs to perform well over the lower bound of what is defined as an interactive frame rate.

- **Create a GPU particle system framework capable of simulating millions of independent particles and add the found method as a feature. What are the conse-
It is important to test the method in a real use case in order to determine how well it performs and what pitfalls it has.

1.4 Delimitations

Artistic control of the particle system is important to take into account, but to keep the thesis work in a reasonable scope, particle editors and tools are not implemented. A custom deferred rendering framework will be built to accommodate the GPU particle system. The focus is to create a particle system capable of delivering visual effects in a convincing manner, e.g. a physically accurate particle simulation is not what the system is for.
In this chapter the techniques and concepts required to implement a GPU particle system are covered. Some of the sections are foundational for computer graphics such as the graphics pipeline and coordinate spaces. It is important to have a good understanding of these foundations when moving on to more advanced techniques such as screen space collisions, mesh material manipulation and particle systems which are imperative in order to reach the goal of this thesis.

2.1 Graphics pipeline

From a programming point of view, the graphics pipeline of the early 2000s consisted of a set of fixed functions called on the CPU to process input data. The fixed function application programming interface (API) allowed users to change the state of the graphics pipeline stages to achieve a desired result. The fixed function pipeline is now obsolete and replaced by fully programmable shader stages, allowing users to upload kernels of code for stages in the graphics pipeline to the GPU. The shader programs provide more flexibility than the fixed functions and requires less communication between the CPU and GPU, which is a common bottleneck for graphics applications.

2.1.1 Overview of Direct3D 11

In Direct3D 11 (D3D11) the graphics pipeline can be described as having two paths, one is for rendering and the other is for computational purposes. The stages of the graphics pipeline are input assembly, vertex, geometry, hull, domain and pixel shaders, lastly the output merger. There's only one stage for the computational path which runs a kernel of code on its resources. Direct3D 11 has quite few allocatable resources, but their properties can be configured in many ways.
The allocatable resources are created by a \textit{device object} which is a hardware independent abstraction of the GPU. The device also needs what is called a \textit{swap chain}, which contains the \textit{back-buffers} on which our scene is rendered before presenting them on the screen. The device also has a rendering \textit{context} which sets state and resources for the graphics pipeline, it is able to set what \textit{render targets}, shaders, \textit{samplers}, vertex and index buffers, \textit{rasterization state} and \textit{blend mode} which is used for an upcoming \textit{draw call}.\cite{11} D3D11 shaders can bind its resources through the device context, and the possibilities depend on which \textit{shader model} the shader is compiled with. The shader model determines what the underlying hardware is capable of, earlier shader models have more limitations to what can be done with shader technology, later shader models do not have as many limitations but are not completely void of them. Shader models among other things determine how many registers and operations that can be used in a shader. Shader model 4.0 introduced compute shaders and supported read-only resources. Shader model 5.0 enables \textit{unordered access views} (UAV) which is a type of resource that has read-and-write capabilities, it also comes with \textit{structured buffers}, which is a shader resource that can contain structures of data and have internal counters.\cite{14} It is important to note that most of the functionality of D3D11 covered in this section also map to at least OpenGL 4.3.\cite{8}

\subsection{2.1.2 Coordinate spaces}

A mesh consists of a set of vertex positions defined in the local coordinate space of the mesh object. To map vertex positions in 3D to a 2D plane, e.g the application window surface, a series of transforms are applied to the vertices. To manipulate the vertices relative to a 3D scene, a world transform, represented by a 4x4 matrix, is applied. A view transform, often referred to as the camera orientation and location, is applied to the world space coordinates of the vertices to move the mesh into the view frustum. A projection matrix represents the camera lens, which determines the shape of the view frustum, and is used to transform the view space coordinates of the vertices into projected space. There are a number of different projections used in graphics applications but for 3D scenes the most common projection is a perspective projection. The perspective projection matrix models a pinhole camera where meshes scale in relation to the view distance. The projected coordinates are mapped to a normalized device coordinate space (NDC) by doing perspective division of the projected vertex coordinates. Finally, a viewport transform is applied to the NDC space coordinates to get the vertex in window coordinates.\cite{13} In D3D11, using the SV\_POSITION semantic on vertex shader position output automatically transforms projected vertex coordinates into window coordinates based on the viewport settings.\cite{12}
2.2 Deferred rendering

Deferred rendering is an alternative to the default way of rendering 3D scenes called forward rendering. The advantage of using deferred rendering over forward rendering is that it’s more computationally efficient for complex scenes with overlapping geometry. The reason for this is that deferred rendering avoids unnecessary computations of geometric fragments which might not even represent a pixel in the final image.

Deferred rendering renders a scene by combining textures filled with the surface properties of a scene. The collection of surface property textures are referred to as the G-buffer. The different textures usually contain the diffuse color, specularity, surface normal and view depth of a scene. In high-end applications, the surface properties are usually tightly packed in the different color channels of the textures, that means a single texture can represent more than a single surface property. [22] Because there are multiple surface properties, a technique called multiple render targets (MRT) is utilized. MRT enables a shader to write to several render targets in a single shader pass. D3D11 supports MRT and without it deferred rendering would need to render the scene again for each surface property texture to fill the G-buffer.

When illuminating a scene with deferred rendering, directional, spot and point light sources are represented as convex geometry such as fullscreen quadrangles, cones and spheres respectively. The light geometry data is uploaded to a lighting accumulation shader which determines lit pixels by using stencil buffer operations. When lit pixels have been determined, a final shader pass takes the information of the G-buffer and applies a lighting model to the lit pixels. Illuminating a scene with deferred rendering has a time complexity of $O(n \cdot p)$ where $p$ is the amount of lit pixels of the current viewport size. In comparison, default forward rendering has a time complexity of $O(n \cdot g)$ where $n$ is the number of lights and $g$ are the potential geometric fragments. [1, pp. 279-283]

The technique described here is one of many ways to implement deferred ren-

---

1 Not referring to Direct3D 11 deferred context rendering.
2 Geometric fragments that will be occluded may be skipped when shading by doing a depth pre-pass and then enabling early depth testing.
There are many specialized implementations and the way the G-buffer is set up and lighting is applied is determined by the requirements of the rendering application. A drawback of using deferred rendering is that there's no support for non-opaque meshes, which needs to be resolved by sorting meshes in back-to-front order and rendering them in a forward rendering fashion.

2.3 Texture sampling

Meshes are often accompanied by textures that are to be mapped on to the surface of a mesh during rendering. In order know what part of the texture goes where, each vertex position in \( \mathbb{R}^3 \) needs a texture coordinate in \( \mathbb{R}^2 \). Texture coordinates will be specified as \((u, v)\) from now on. Texture coordinates are linearly interpolated between vertex positions in the shader. Repeating textures can be achieved by having texture coordinates \(|u| > 1\) or \(|v| > 1\). Texture coordinates provides a resolution independent way of accessing values from the texture.

In order to access values from the texture, it is treated as a signal; signal processing techniques apply. The texture is treated as the input signal and sampled at discrete texel locations. The texture coordinates \((u, v)\) determine what texel to sample from. The look of the output signal depends on how the samples are interpolated. Texture samplers exist in modern GPU hardware. Hardware texture samplers can use different state to allow repeating, mirrored or clamped texture mapping, but also specify how samples will be interpolated. There are several ways of interpolating between samples which will be covered in the following sections.

2.3.1 Nearest neighbor

Nearest neighbor sampling also known as point sampling retrieves the nearest texel at the given sample point. The nearest neighbor sampling does not perform any interpolation when trying to sample in between two texels but rather chooses one or the other based on distance. The nearest neighbor sampling can be used for retaining a pixelated look when texture mapping or to avoid getting interpolated values when encoding computations into textures. Nearest neighbor samplers exists in hardware.

2.3.2 Linear

Linear filtering retrieves the 2 nearest texels for the given sample point if the sample point is between two texels, it then chooses a weighted average of the 2 texels based on the sample point. Linear filtering can be extended to 2D which gives a bilinear filtering, which chooses a weighted average between the 4 nearest texels to the sample point. Linear sampling may result in artifacts, especially

---

3In D3D11 raw texture values can be accessed without a sampler by using a RWTexure2D resource view.
if there are discontinuities in the texture. Linear filtering results in a smooth gradient between texel values and linear samplers exist in hardware.

2.3.3 Cubic with B-spline weights

In the case of linear filtering the smoothing may cause high frequency areas of the texture to lose detail when mapped to a surface, especially if there is magnification. Magnification occurs when one texel of the texture covers more than one pixel on the screen. The loss of detail can be mitigated by choosing another sample curve. A curve can be represented by a polynomial function. Curves can be approximated by using a Taylor expansion, but is problematic since the approximation converges slowly when the distance from the origin and polynomial order increases, this becomes computationally expensive. Instead of having a single high order polynomial to represent the curve, curve segments of lower polynomial order can be fitted after each other. There are several ways of accomplishing curve fitting and B-splines is one of them. A B-spline consist of a set of basis functions of some degree. A linear combination of the basis functions and weights approximate the curve.

\[ p(t) = \sum_i c_i b_i(t) \]  

(2.1)

Each of these basis functions along with its weight corresponds to a unit spaced control point, often called a knot. The summation formula gives the position along the curve based on the argument \( t \) where \( i \) is the current control point, see equation 2.1. The piecewise polynomial function for a B-spline of degree \( n \) cannot be computed directly but must be computed recursively using De Boor’s algorithm. The piecewise polynomial for a uniform cubic B-spline can be seen in equation 2.2. A uniform cubic B-spline is \( C^2 \) continuous and can therefore give a smooth transition between curve segments.

\[
B_{i=0,3}(t) = \\
\begin{cases} 
(t + 2)^3, & -2 \leq t < -1 \\
-3(t + 1)^3 + 3(t + 1)^2 + 3(t + 1) + 1, & -1 \leq t < 0 \\
3t^3 - 6t^2 + 4, & 0 \leq t < 1 \\
-(t - 1)^3 + 3(t - 1)^2 - 3(t - 1) + 1, & 1 \leq t \leq 2 \\
0, & \text{otherwise}
\end{cases} 
\]  

(2.2)

Cubic B-splines may be used for filtering in the sampling of a texture to retain some detail when magnifying, see figures 2.2 and 2.3 for a comparison between bilinear and bicubic filtering. The filtering can be extended to higher dimensions easily. The naive way of implementing this filtering requires 16 texture lookups for a 2D texture, however it can be optimized to be computed in only 4 texture lookups using a hardware bilinear sampler. This is due to the fact that cubic B-spline filtering is a natural extension to linear filtering. [20]
2.4 Billboards

Billboarding is used to orient a polygon to face a certain direction. Billboarding can be used for a number of things such as rendering text, sprites, vegetation and particles. They may also be used for level of detail to gain performance; complex meshes can be replaced with a billboard when the mesh covers a small enough screen area, this type of billboard is called an impostor. A billboard can be represented by either a textured quadrangle or triangle. Using quadrangles is perhaps more intuitive than mapping a rectangular texture onto a triangle correctly. There are 5 ways of orienting a billboard based on the context it is used in, described in the following sections. [1, pp. 446-457]

2.4.1 World oriented

The billboard does not rotate with respect to the camera, but rather has a fixed orientation in world space. To achieve this, nothing but the world, view and projection transforms are applied to the vertices of the billboard polygon.

2.4.2 View point oriented

The billboard always face the view point, the position of the camera. To achieve this, an orthonormal basis representing the orientation needs to be constructed. The orthonormal basis vectors for a view point oriented billboard can be computed in a vertex shader. The desired normal direction of the of the billboard is found by $\hat{z} = \frac{vp-center}{|vp-center|}$ which is the direction between the view point and the billboard origin. A cross product between the view direction and the normal di-
rection of the billboard gives an up vector for the orthonormal basis, \( \hat{y} = \hat{v} \times \hat{z} \) where \( \hat{v} \) is the view direction. The last vector for the orthonormal base is found by taking the cross product between the billboard normal direction and the up direction \( \hat{x} = \hat{z} \times \hat{y} \). This orthonormal basis can be used as a transform to orient the vertices of the billboard polygon.

### 2.4.3 View point oriented, axial

The billboard always face the view position of the camera, but rotation is restricted to a single axis in world space. To restrict the rotation most of the procedure of view point orienting a billboard is done. However, the component of the restricted axis, e.g the Y-axis is set to zero when finding the normal direction of the billboard.

### 2.4.4 View plane oriented

This billboard technique always face the view plane of the camera, it is screen aligned. This type of billboarding is achieved by finding the orthonormal basis vectors in a similar way as with the view point oriented billboard. Instead of choosing the vector between the view point and the billboard origin as the normal direction of the billboard, the negated view direction is chosen.

### 2.4.5 View plane oriented, axial

The billboard always face the view plane of the camera, but rotation is restricted to a single axis. This is done in the same manner as with the view point oriented axial billboard, but using view plane oriented billboarding instead.

### 2.5 Particle systems

There are many ways to structure a particle system. Some of the modern particle systems that have been implemented on the GPU and are used in the game industry are presented by Thomas [6], Rockenbeck [18] and Sanisalo [19], they all have similar features but all of them differ in their implementation and structure based on how the systems are used. In this thesis work, a particle system is assumed to have one or more particle emitters. Each particle emitter spawns individual particles based on the emitter properties. The emitter controls the lifetime and initial behavior of individual particles. Common emitter properties can be the shape in which to spawn individual particles e.g a sphere, line or any other geometric shape, color, lifetime, velocity, changes over time etc. They basically define the visuals of the particles spawned from that emitter. Since particle systems can contain several particle emitters, complex effects may be achieved by combining emitters in the desired way. There are two ways of propagating particles through space. Individual particles can be either *stateless* or *stateful*, both are described in the sections 2.5.1 and 2.5.2 respectively.
2.5.1 Stateless particles

In a stateless particle system, a particle’s movement is defined by a closed form function which is entirely controlled by some input value, usually the current lifetime of the particle. Stateless particle systems can be used for effects that do not interact with the environment. They may be used for effects that emit particles in shapes or curves that are easily expressed mathematically. Stateless particles have limited use in computer games; it can be hard to find and tweak a closed form function that propagates the particle in a desired way.

2.5.2 Stateful particles

Stateful particles on the other hand do not have the limitation of having to deal with closed form functions. [9] A stateful particle has its initial position explicitly defined and is propagated with the equations of motion by integrating the position with respect to the velocity and acceleration of the particle. The particle propagation is essentially an ordinary differential equation (ODE) since the particle position is found with the help of its derivatives with respect to time.

Because the particle propagation is an ODE it needs to be integrated numerically. There are several methods of numerical integration that may be chosen based on the numerical stability that is required for the integration. For instance, using Euler integration may not be enough due to floating-point precision errors. In cases where a particle’s movement depends on other particles, or where particles have a long life time, the floating-point errors would add up quickly and the simulation would become increasingly unstable. Other methods of numerical integration such as Verlet integration or some order of Runge-Kutta integration can be used to provide increased numerical stability, possibly in exchange of some performance. [10, ch. 5]

Particle collision response is important to get a sense of mass. In order to elicit a collision response from the geometry of the scene the normal of the surface of the collision point needs to be known, as well as the velocity of the particle. By taking the component vectors of the particle velocity before a collision, the equations to get the components are seen in equation 2.3. The collision response is different based on the surface material of the collision point, to simulate this two other parameters are added, namely dampening and restitution. The particle velocity after the collision is found by substituting the dampening and restitution parameters into equation 2.4 where $\alpha$ is the dampening, $\beta$ the restitution and $N$ is the surface normal. [9]

\begin{equation}
\begin{cases}
V_N = (V \cdot N) \cdot V \\
V_T = V - V_n
\end{cases}
\end{equation} \hspace{1cm} (2.3)

\begin{equation}
V_{After} = (1 - \alpha) \cdot V_T - \beta \cdot V_N
\end{equation} \hspace{1cm} (2.4)
2.5.3 Particle representation

Individual particles are often represented as a textured billboarded quadrangle. Because particles mostly come in large quantities it would sink performance to represent each particle as a complex mesh, even though such methods exists as well. The particles are usually small with a short lifespan which makes it hard to notice that they are in fact billboarded quadrangles.

Particle systems can have a local coordinate system of which particle emitters exist in. A useful way of modeling particle systems is by looking at them as a hierarchy or graph where the particle system is the root node. If the particle system and its emitters are represented as a graph, complex movement of the particle system and its emitters can be achieved. To produce visually appealing effects, particle emitters can use noise algorithms to introduce some form of randomness to the particle system. The noise values can be used to offset the initial behavior of particles from the particle emitter.

Particles can either be semi-transparent or fully opaque and be textured. Translucent particles require sorting particle depth in front-to-back order to be rendered properly due to depth-testing limitations.

2.6 Screen space collisions

Screen space collision detection is a simple method of doing collision detection by comparing an object’s view depth and the view depth of other scene objects. Because the depth information of the rendered scene is stored in the depth buffer it can be sampled at any time in D3D11. To get the object view depth, the depth buffer is sampled at the object’s screen space position. A collision has occurred if the object’s view depth is larger than the scene depth at the same position. However, with screen space collisions there’s no information about the actual thickness of scene objects and must use an artificial thickness value. Without an artificial thickness value, an object in the depth buffer could be interpreted as being infinitely thick passed the depth value, resulting in false positives for collisions. The artificial thickness value is highly scene dependent and need to be tweaked until the collisions behave in a plausible way. [21] In the case of particle systems, the sampling of the object’s view depth can be approximated for smaller particles by sampling the depth buffer at the particle’s center, larger particles would need to sample the depth buffer at more points to be convincing.

The first step of collision checking using this method is to check the condition $z_{\text{scene}} < z_{\text{particle}}$. This would cause particles far behind the scene depth to still collide because there is no thickness information involved. The extended collision check is to add the artificial thickness value, $z_{\text{scene}} < z_{\text{particle}} \land z_{\text{particle}} < z_{\text{scene}} + \epsilon$ where $\epsilon$ is the artificial thickness value. As mentioned earlier, the artificial thickness value is scene dependent, e.g a larger world scale would require a larger thickness value for collisions to look convincing. This allows for collision checking in such a way that the particle depth is past the scene depth, but not farther than the thickness of an object. If the particle depth is passed the scene depth and thickness value, the particle is behind the object.
2.7 Mesh material manipulation

A method of mesh material manipulation as the result of a particle collision, or painting, was proposed by Drone [4]. The method checks for collisions between the world space positions of meshes and particles. If there was a collision between a particle and a mesh then paint needs to be applied to that mesh.

In order to know where on a mesh to paint, a mapping between the texture coordinates and world space positions of the mesh is needed. This mapping is obtained by using a render-to-texture approach where the world space positions of the mesh is written into its texture coordinate layout. The world space positions written into the texture space of the mesh can be seen in figure 2.5. In more detail, this world space position texture can be obtained by the following steps:

1. Setting the viewport size to cover the texture dimensions.
2. Rendering the mesh, providing vertex positions and texture coordinates.
3. Making sure texture coordinates are in the range \([0, 1]\).
4. Treat texture coordinates as the vertex position at vertex shader output.
5. Pass the world space position of the mesh to the pixel shader.
6. Write the world space position to texture at pixel shader output.

The world space positions for collided particles are written into another texture during rendering of the particle system, see figures 2.4. In modern graphics APIs this texture could be created with MRT or read-write buffers. Note that this texture has nothing to do with the representation of the particle system itself; only particles that have collided somewhere in the scene are written into this texture.

When both world space position textures have been created, another shader pass, which also uses the render-to-texture approach, applies the painting. The mesh position texture and particle position textures are passed as SRVs and another texture, the paint texture, is bound as the RTV for the shader. Both of the position textures are looked up in the pixel shader, each position of the mesh is compared with the position of each collided particle. If the compared positions are near enough, paint is output to the RTV texture. Because of the mapping between texture coordinates and world space position the paint is implicitly written into the correct location in the mesh's texture space.

The method works under the assumption that each instance of a mesh has its own position and paint textures. The values of the world space positions can exceed the range of a regular texture with 8-bits per channel and may need to be encoded so that they fit. Alternatively, a 16-bit or 32-bit per channel floating-point texture can be used for positions depending on the scale of the scene. The dimensions of the textures depends on the amount of detail needed for painting.

The cost of iterating through each particle position for each mesh position is high and requires dynamic branching in the pixel shader. The method amortizes
the cost over time. It does this by iterating over a constant number $n$ of particle positions for one frame and the next $n$ of particle positions the next frame.

Lastly, in order to have a persistent paint, the results from the previous frame needs to be stored in another texture of the same size and format. The previous frame's paint result can be looked up and reapplied so that no information is lost, this is accomplished by ping-ponging between the paint textures, which is a common technique in post processing effects.

2.8 Blending

Blending values of two different sources in computer graphics is essential to allow for certain effects. Most often the two different sources are called source and destination, the source referring to what is being computed and the destination to what has already been computed. The RGB and alpha channel values of both the source and destination are blended together in a way determined by the set blending operation and blending functions. There are many blend modes to choose from and in D3D11 the current blend mode is set as state for the output merger stage. D3D11 allows for different blend modes on different RTVs as well as a blend function and the blend operations. The usual blending operations are add, subtract, reverse subtract, min and max. The blend functions available for both source and destination values are one, zero, color, one minus color, alpha and one minus alpha. The blend equation for the add operation can be seen in equation 2.5, this shows the general case. In modern graphics APIs the applied blending functions may be different for the color and alpha channels. The blend equations for the other blend operators look similar but are not covered in this
section. [15] [16]

\[ \text{DEST} = \text{SRC}_{\text{func}} \ast \text{SRC} + \text{DEST}_{\text{func}} \ast \text{DEST} \] (2.5)

2.9 Noise

Noise algorithms are utilized to introduce randomness to certain elements in computer graphics. Some use cases are normal and vertex displacements, creating or adding detail to textures. By procedurally adding detail to textures we’re able to use smaller textures and offload some texture memory and bandwidth usage in exchange for ALU operations. GPUs are so powerful that it is possible to generate noise in real-time applications. There are many different kinds of noise algorithms but in this section only one will be covered that works well on the GPU, namely Perlin noise. The reasoning for covering only Perlin noise is because of its properties: it is pseudo-random, meaning it gives the same noise value for a given hash; it has no discontinuities, except for some artifacts; it is easy to extend into higher dimensions.

Perlin noise was authored by Ken Perlin, hence the name. It is a gradient noise in N dimensions, which means that gradients are spaced at regular intervals as in a lattice. The noise algorithm is sampled with a point in N dimensions. Not unlike bilinear filtering, for N = 2 the four nearest interval points are chosen. At each interval point the gradient value is 0 and the gradients for each contributing interval point is interpolated to yield the final noise value. That essentially means that the integer part of the point is the lattice point which in turn chooses which gradient to use, the fractional part is used to determine how much influence a gradient has when interpolating between other gradients.

By sampling the noise function at different frequencies, usually multiples of the start frequency (octaves) a turbulent noise may be achieved which exhibits fractal patterns. Such a turbulent noise is usually referred to as fractional Brownian motion, see image 2.6. Fractional Brownian motion may be used to create procedurally generated graphics such as terrain, clouds etc. [5]

An upgrade from Perlin noise is simplex noise which removes some of the artifacts of Perlin noise and instead of interpolating between gradients only a few summations are needed. Simplex noise is not covered in this section but is explained well by Gustavsson [7].

2.10 Texture splatting

Texture splatting is a special case of multi-texturing. It is a technique used to blend two textures together based on weights from a third texture. The weight textures are often represented as a grayscale bitmap where specific ranges of brightness determine the blended value. An example could be blending grass and dirt textures on a terrain based on a weight texture.

In the case of texture splatting on meshes that occupy a large screen space area there is magnification of the weight texture. The magnification smooths out
2.10 Texture splatting

Figure 2.6: Perlin noise sampled at different frequencies resulting in fractional Brownian motion.

detail when using a bilinear filter as mentioned in section 2.3. To preserve detail of the texture splatting a bicubic filter could be used, and to add detail a noise function may be blended together with the weight texture. [3]

An example of the results of using texture splatting is seen in figure 2.10 where the figure 2.7 is the first input texture and the figure 2.8 is the second input texture, the weight texture is seen in figure 2.9. The final image is obtained by sampling the textures and linearly interpolating between the RGB texture values based on the scalar value obtained from the weight texture \( \text{Result} = \text{Image}_0 \cdot \text{Weight} + \text{Image}_1 \cdot (1 - \text{Weight}) \), where \( \text{Weight} \in [0, 1] \).
The particle system is written in C++11 along with the Direct3D 11 API. Texture bitmaps are loaded with the DirectXTex library and models are loaded with Open Asset Importer library. The choice of C++11 is based on previous experience and C++ being the industry standard for low-level game development. The reasoning behind using Direct3D 11 is that it targets a large scope of hardware, but any other graphics API with the same features could be used as well. Other interesting choices of graphics APIs would be OpenGL 4.5, Direct3D 12 or Vulkan. The most similar graphics API to Direct3D 11 of those mentioned is OpenGL 4.5. As for Direct3D 12 or Vulkan they require more setup and management, choosing either would increase the time needed to get the project up and running, which would detract from the focus of this thesis work. The implementation is based on the GPU particle system presented by Thomas [6] as well as the particle geometry interaction presented by Drone [4]. Other features have been added to make the particle system more flexible in terms of use.

3.1 Rendering framework overview

The underlying framework consists of a collection of classes. The base of the entire framework is a device class which holds the Direct3D back-buffer and the and the immediate context, the device class is needed anywhere in the code where GPU resources are allocated. It is essentially a wrapper with convenience methods for getting and setting the state of the current context. Several resource wrappers are available to create constant buffers, raw buffers, textures, samplers and blend modes.
3.1.1 Deferred geometry pass

The geometry pass of the deferred renderer allocates RTVs for the surface properties that will be contained in the G-buffer, as well as a depth-stencil target which is separate from the G-buffer. The shader program for the geometry pass consists of vertex and pixel shader stages. The surface properties gathered in this particular implementation are few, only consisting of the diffuse color, specularity and the view space normals. The geometry pass could easily be extended to support more surface properties at a performance cost. Other implementations may provide a position target for the view space positions of the meshes but is in this case unnecessary since position can be reconstructed from the depth buffer, it is essentially a trade off between texture bandwidth usage and ALU operations. At rendering time, all of the G-buffer targets are bound as RTVs and the depth-stencil target is bound as the current depth-stencil view (DSV) in the device context. Each mesh is rendered with the shader program, binding the mesh specific textures as SRVs. The mesh surface properties are written into the G-buffer. After all the meshes are rendered, the render targets are unbound so that they can be reused as SRVs in other shaders.

Diffuse target The diffuse color G-buffer target uses R11G11B10_FLOAT for the buffer, SRV and RTV. The reason for choosing this particular format is that there is no need for an alpha channel in this case, deferred rendering does not handle non-opaque materials anyway. A format of R8G8B8A8_UNORM could be used instead.

View space normals and specularity target The view space normals and specularity G-buffer target uses R8G8B8A8_SNORM for the buffer, SRV and RTV. The reasoning for choosing this particular format is that there is no need for more precision and negative values are expected. The specularity of a material is interpreted as a scalar value between 0 and 1, but can later be multiplied by a constant to be used as a specular power for the used lighting model in the deferred lighting pass.

Depth-stencil target The depth and stencil target uses R24G8_TYPELESS for the buffer, R24_UNORM_X8_TYPELESS for the SRV and D24_UNORM_S8_UINT for the DSV. The reasoning behind the different formats is that the some resource views do not support a certain format.

3.1.2 Deferred lighting pass

The lighting pass of the deferred renderer binds the contents of the G-buffer as SRVs to a lighting shader. The lighting shader program consists of vertex and pixel shader stages. In this particular implementation point lights are not implemented, although the lighting pass could be extended to support them. Instead, a constant buffer of directional light data is uploaded to the lighting shader. The lighting pixel shader loops through each of the directional lights and computes
the light contribution at each pixel. The results are written to another offscreen RTV which can be used later on when adding post-processing effects. The lighting model used in this particular implementation is Blinn-Phong, but could be replaced with any other such as Cook-Torrence or Oren-Nayar to give a more physically based lighting. A directional light is defined as a struct with a direction and intensity and is applied to the whole scene, since it is thought of as being infinitely far away. The constant buffer containing the directional light structs are limited to 8 directional lights in total.

### 3.2 Particle system framework

The particle system framework consists of a collection of C++ classes. The ParticleSystem class manage initialization, emission and rendering of a single particle system, each ParticleSystem defines a maximum amount of particles available for that particular system. The ParticleSystem is able to add derived objects of the abstract ParticleEmitterBase class that will use the available particles of the ParticleSystem. The ParticleEmitterBase class has setters for the general behavior of a particle emitter, derived classes from ParticleEmitterBase define specific types of emitters, such as the emission shape. For example, the ParticleConeEmitter class has setters for specific behavior of the cone shaped emitter such as rotation, angle and height. Each derived class of ParticleEmitterBase has a specific shader which is stored in a ParticleSystemShaders class, which acts as a shader cache for particle systems. Since there can be many emitters of the same type and therefore using the same shader, there is no need to recompile shaders for the same type of emitter with a shader cache.

At instantiation of a ParticleSystem an enum ParticleSystemBlendMode is passed to its constructor, there are three different blend modes implemented, additive, multiplicative, alpha blending and the option to not use one at all. These blend modes adds to the visual aspect of the particle rendering where a certain effect may only be achieved with a certain blend mode. The ParticleSystem also needs to attach an instance of the ParticleSystemSharedResources class in order to be able to render correctly. The ParticleSystemSharedResources class contains the render targets and blend modes which can be used when rendering a ParticleSystem. In order to apply textures to the particles a ParticleSystemTextureArray object can be attached to the ParticleSystem which creates a D3D11 texture array containing texture data. Using a texture array, emitters using different textures can still use the same resource but using different indices into the texture array. Textures can be added and loaded by supplying a path to the texture file and an associated name for the texture. When adding a ParticleEmitterBase object to the ParticleSystem instance the texture name is also passed as a parameter. The attached ParticleSystemTextureArray object can be used to query the texture name and return an index into the texture array which can be used by the particle emitter and in the particle rendering.
3.3 Particle system shaders

![Diagram of particle system shaders](image)

**Figure 3.1:** A diagram showing the components of the particle system on the GPU. The arrows represent how the components read and write data, data can be transferred from the base to the head of the arrow. Some components are both read and write enabled.

The particle system from the GPU point of view consists of three buffers of data. The first buffer is the particle buffer, which is a chunk of memory on the GPU containing particle structs, it is represented by a D3D11 RWStructuredBuffer<Particle>. The second buffer is the dead list which keeps indices into the particle buffer of particles who has exceeded their lifetime, represented by a D3D11 AppendStructuredBuffer<uint> and ConsumeStructuredBuffer<uint>. The third and final buffer is the render list which keeps indices into the particle buffer of particles who has lived through a frame and should be rendered, represented by a D3D11 RWStructuredBuffer<uint>. The initialization, emission, updating and rendering of particles are done by running shader programs for each respective step from the ParticleSystem class. Both the dead list and render list has hidden counters which can be fetched on the CPU, these hidden counters can be incremented and decremented by either invoking the methods Append and Consume or IncrementCounter and DecrementCounter depending on the buffer type, this enables other shaders to check the length of the lists. Internally, the lists are still raw buffers of memory but the counter allows treating them as a list data structure. There is yet another buffer of data that is bound during the update step which is for indirect draw arguments, like the hidden counters the contents of this buffer can be fetched on the CPU allowing us to initiate draw calls with a variable number of vertices. It is important to note that these buffers...
of data are classified as UAVs if the shader both reads and writes to the buffer, otherwise they are bound as SRVs. The resources using the RW prefix as well as the append, consume and counter buffers are UAV resources. For an overview of the system on the GPU and the relationship between the components, see figure 3.1. A keen eye may have noticed that there is no sorting of particles based on depth in this particular implementation, partly because writing a GPU sort is much work and by using certain blend modes it is hard to notice the render order of particles.

### 3.3.1 Initialization step

The initialization shader program consists of a compute shader stage, it is dispatched once at particle system creation with the maximum number (1024) of threads per thread group. The initialization shader places all of the indices of the particle buffer in the dead list by invoking the append method of the AppendStructuredBuffer<uint>. Every particle is presumed to have exceeded its lifetime at particle system creation, this gives the attached particle emitters available particles at the beginning of a particle simulation. The dead list counter enables the emitter shaders to check if there is any available particles at emission time.

### 3.3.2 Emission step

The emitter shader programs each consist of a compute shader stage, they are dispatched as the first step of the simulation cycle with the maximum number (1024) of threads per group. Each specific emitter maps its data to D3D11 constant buffers which are uploaded to the specific emitter shader. There is a separation between specific emitter data and general emitter data which is uploaded as separate constant buffers. The thread ID of the compute shader is compared with the dead list counter, if the thread ID is less than the dead list counter the emitter can emit one particle. The initial values of each emitted particle is set by reading from the emitter shader data. The initial world positions of the particles are set by the specific emitter data, offset by GPU noise to provide some randomness. If a particle system has more than one emitter they are invoked in the order they were added. If by chance there is no available particles at emission, the thread ID comparison mentioned earlier will prevent the emitter shader from consuming data outside the bounds of the dead list.

```cpp
struct EmitterData {
    float3 position;
    float lifeTime;
    float3 velocity;
    float spawnTime;
    float3 acceleration;
    float maxEmission;
    float textureIndex;
    float emitterIndex;
};
```
float speed;
float isVolume;
};

Listing 3.1: The general emitter data struct in HLSL.

struct ConeEmitterData {
  float4x4 rotation;
  float height;
  float angle;
};

Listing 3.2: The specific emitter data struct in HLSL for a cone shaped emitter.

3.3.3 Updating step

The update shader program consist of a compute shader stage, it is invoked as the second step of the simulation cycle. Compared to the other simulation steps, this step is using a smaller number (128) of threads per group. The reason for using less threads per group is that the shader has quite a bit of dynamic branching, as each thread group needs to wait for all the threads in the group to finish [23]. The update compute shader is dispatched for as many thread groups that are needed (with the given threads per group) to cover the whole particle buffer. The thread ID indexes the particle buffer, if the indexed particle is still alive it will be updated.

The G-buffer target for the surface normals and the depth stencil view from the deferred geometry pass are bound as SRVs. For checking screen space collisions, the world space position is transformed into normalized device coordinate space, it is then scaled and biased so that it can be used as a texture coordinate for a texture lookup for the depth buffer. A comparison is done between the view depth of the particle and the linearized depth from the depth buffer, see section 2.6. If there is a collision, the G-buffer normal target is sampled with the same texture coordinate to get the surface normal at the collision point. The surface normal is used for the collision response, see section 2.5. The particles may also be put in a sleep state after colliding for an amount of frames defined by the user. A particle set in the sleep state is not propagated nor checked for collisions and remain dormant for the rest of its lifetime, the sleep state flag is seen in listing 3.4.

The position of the particle is propagated with the equations of motion, in this case Euler integration is used because performance is prioritized over numerical stability. The lifetime of the particle is reduced by the elapsed frame time. Particles that has a depth that exceeds the scene depth as well as the scene thickness value are set in a culled state. If the particle’s lifetime is exceeded, it is put back into the dead list, otherwise its put into the render list and the hidden counter of the render list is incremented, if the particle is in a culled state it is not put in the render list. The indirect draw arguments vertex counter is also increased by a
number of 6, which is the number of vertices needed to represent a non-indexed quadrangle. In the beginning of the update step the indirect draw arguments are reset.

```c
struct Particle {
    float3 position;
    float3 velocity;
    float3 acceleration;
    float lifeTime;
    float age;
    float textureIndex;
    float collisions;
    float culling;
    float painting;
    float sleeping;
    float emitterIndex;
};
```

*Listing 3.3: The particle data struct in HLSL.*

### 3.3.4 Rendering step

The render shader program consist of vertex and pixel shader stages, it is invoked as the third and final step of the simulation cycle. The indirect draw arguments from the update step have been fetched on the CPU and has been used as the argument for the draw call that invokes the rendering shader program.

The vertex shader uses the D3D11 built-in variable VertexID which represents a unique ID for the current vertex of the draw call. Since each particle is represented by a quadrangle made of 2 triangles it is known that 6 vertices are expected for a single particle. The VertexID is divided by 6 to get the index into the particle buffer (which is bound as an SRV), and using a modulo operator gives the current quadrangle vertex. The quadrangle is then billboarded to face the viewpoint, and manipulated based on the size given by the particle emitter. Each particle emitter also supplies variables that change over a particle's lifetime, most of these variables are interpolated in the vertex shader of this step, see listing 3.4, note that this data is also available in the update step. Most of the variables that change over a particle's lifetime consists of 4 values that are interpolated with B-splines. The color tint of the pixel, size and rotation are interpolated variables. An important note about the rotation is that it is not applied directly to the billboard, but rather to the texture coordinates of the quadrangle representing the particle. There is also a stretching factor which transforms the world velocity to view space velocity by applying the view transform, then uses a scale factor to stretch the billboard. The role of the stretching effect is essentially motion blur.

The pixel shader looks up the texture from the particle system texture array with the hardware interpolated texture coordinate of the quadrangle and a texture array index set by the emitter at particle emission. This shader uses MRT to write the particle color result to one RTV and the particle collisions in
screen space into another RTV. The color RTV is the output from the deferred shading and the collision RTV is a texture of the same size and uses the format R16G16B16A16_FLOAT. The collision RTV is only written to if the rendered particle is set in a collided state from the update step, and if collisions are enabled for the particle emitter. The collision RTV has the color that should be used for painting in its RGB channels and the view depth in the Alpha channel, these values will be needed when painting meshes at a later stage.

```hlsll
struct VariableParticleEmitterData {
    float4 colorPoint0;
    float4 colorPoint1;
    float4 colorPoint2;
    float4 colorPoint3;
    float4 sizePoint01;
    float4 sizePoint23;
    float4 rotationPoints;
    float collisionsEnabled;
    float collisionDampening;
    float restitutionFactor;
    float paintingEnabled;
    float4 paintColor;
    float sleepingEnabled;
    float sleepingCollisionFrames;
    float stretchingEnabled;
    float stretchFactor;
};
```

*Listing 3.4: The variable emitter data struct in HLSL.*

### 3.4 Particle painting

In comparison to Drone’s method, described in section 2.7, this method does not write the world positions of the mesh to its own separate texture. It removes the loop checking of particle intersections in the pixel shader of the paint application shader program. Instead of working in world space positions, the method works entirely with screen space positions. The particle collision position texture is replaced by a fullscreen texture, containing the screen space positions of particles that are supposed to paint objects in the scene. This screen space collision position texture is written during the rendering of the particle system, explained in section 3.3.4.

#### 3.4.1 Overview and setup

In order to paint scene meshes as a result of a particle collision two paint textures are allocated for each mesh that is subject to painting. The textures are of resolution 512x512 pixels, in the format R8G8B8A8_UNORM. Any texture format or
resolution could be used but this set up provides enough colors and detail for the purpose of this thesis work. Larger texture resolutions would increase the amount of render operations needed to paint a mesh, similarly using a floating-point format for the texture e.g R16G16B16A16_FLOAT would allow for glowing paint using a bloom filter but would occupy more texture memory and increase bandwidth usage.

A maximum number of textures are allocated at the start of the program, the textures are created in pairs and each mesh that is to be painted uses a free texture pair. If no free texture pairs are available then no paint will be applied to that mesh. Textures are only dedicated to meshes if there has been an intersection with a "paint enabled" particle system. The intersection is checked by having an axis-aligned bounding box (AABB) for all meshes and particle systems in the scene. If there is an intersection between a mesh and a particle system, a paint texture pair is associated with that mesh. Each mesh has its own unique ID that is set when loading the mesh asset, this ID is used as a lookup to a dictionary for the dedicated paint texture pair. A time stamp for each texture pair is also added, so that paint texture pairs that have been dedicated for a mesh earlier can be reassigned to another mesh if the user desires that behavior.

The set of possible intersections between meshes and particle systems is reduced by performing a view frustum culling on the scene. Only dedicating texture pairs when there is a clear intersection between a particle system and a mesh allows the method to work in larger scenes without prematurely knowing what meshes could be subject to painting. The reasoning for using a pair of textures for each mesh is that one of the goals of the method is having a paint that is persistent from frame to frame. By keeping the rendered results from a previous frame in one texture and drawing the paint results of the current frame into the other texture, it is possible to have a persistent paint. If there is no new paint applied to the mesh during a frame, the results from the previous frame are rewritten. It is essentially using a ping-pong approach by swapping the two paint textures, using one as a RTV and the other as a SRV for the paint shader. The method iterates over all intersected meshes and renders them with the paint shader only if they have dedicated texture pairs, for details about the paint shader see section 3.4.2.

In order to be able to draw directly onto the texture, a quadrangle covering the entire viewport is rendered, and the viewport is set to the dimensions of the paint texture, which in this case is 512x512 in width and height respectively. One of the textures in the paint texture pair is bound as an RTV for the shader and the other as the SRV. The fullscreen texture of the particle collisions is also bound as an SRV so that it can be looked up. The paint texture output is then fed back to the deferred geometry pass and used as a texture SRV when rendering the corresponding mesh the next frame, see figure 3.2 for an overview of the painting procedure. Because the limitations on paint texture resolutions for larger scenes, the deferred geometry pass samples the paint texture using a bicubic weighted B-spline filtering, applying some octaves of simplex noise at high frequency portions of the paint textures before writing it to any of the G-buffer targets. The possibility of choosing which G-buffer target to apply the paint to
creates an opportunity to paint specularity, bump maps, diffuse color and other surface properties.

### 3.4.2 Paint shader program

![Diagram of the painting procedure from vertex shader input to pixel shader output.](image)

**Figure 3.2:** The painting procedure from vertex shader input to pixel shader output.

The painting is done using a single shader program with vertex and pixel shader stages. Because the particle collisions have been written into the screen space of the fullscreen texture at an earlier stage, the screen space position of the rendered mesh can be used to lookup the fullscreen texture to check for collisions.
If there is a collision, the paint color, which was also written into the fullscreen texture of particle collisions, is written to the current paint texture.

**Vertex shader** In order to get the vertex position in screen space, the view and projection transforms are applied. However, in order to know which position of the mesh to paint, a mapping from projected coordinates to texture coordinates is needed. To realize this mapping, a swap between the vertex position and texture coordinate is performed at vertex shader output in the same way as mentioned in section 2.7, interpreting the texture coordinate as the vertex position in the pixel shader. Since the projected position is now passed to the pixel shader without the SV_POSITION semantic, the perspective divide and viewport transform needs to be applied manually in the pixel shader to get a screen space position. Before passing the texture coordinate off to the pixel shader, it is important that texture coordinates are scaled and biased so that they exist in the range [0, 1] in each dimension. To accomplish the scaling and biasing of the texture coordinate, a UV-transform is constructed when loading the mesh asset which is passed to the paint shader. The texture coordinate is passed to the pixel shader with the SV_POSITION semantic so that perspective divide and viewport transforms are applied automatically between shader stages, the Z-coordinate is always set to zero since the draw surface is meant to be at the near plane. The linear depth of the vertex position is also passed along to the pixel shader, to get the linear depth the projection transform is left out, choosing the z component.

**Pixel shader** With the implicit mapping from projected space coordinates to texture coordinates, intersections between the mesh and the particle collision is done with a single texture lookup (in comparison to Drone’s method which at this stage iterated through $n$ texels of the particle collisions position texture for each painted object). First, the projected space coordinate needs to be converted to a value that can be used to lookup the fullscreen texture of particle collisions. This is accomplished by first checking if the projected coordinate is within the $[-1, 1]$ range in the x and y components, if the position is outside this range, it is not on screen and the value from the previous paint texture can be output directly. The next step is performing a perspective divide of the projected space coordinate’s x and y components which essentially maps the coordinate to the near plane in the range $[-1, 1]$, this coordinate is then scaled and biased to exist in the range $[0, 1]$ which is the screen space position without the viewport transform. This screen space position is then used to lookup the fullscreen particle collision texture. If the lookup returns a non-zero value, there has been an intersection, and painting is applied with the color value written into the fullscreen particle collision texture. Since the method works in screen space coordinates the intersection is only compared in a 2D plane, for this reason the view depth written into the fullscreen particle collision texture is compared to the view depth of the rendered mesh. If the depth values are close enough, which is a user defined value, the paint is output.
The time complexity of the paint method is tested by rendering a scene consisting of evenly spaced cube meshes. Each cube consists of 8 vertices and 36 indices, each allocating two 512x512 textures for paint application. The view frustum is fixed so that all of the cubes are within the frustum during the test. To avoid any randomness introduced by particle systems they are not used at all for the paint performance measurements. Instead, we assume that there is a particle collision at each screen space position of the mesh, always painting them completely. The only thing determining if paint should be applied to a mesh is the intersection between an AABB representing an imaginative particle system and the AABB of the mesh. This gives a good estimation on the time complexity of the method as the amount of paintable meshes increases.

It is difficult to get reliable tests of particle system rendering, all the particles are transient and there is a possibility of overdraw affecting the fillrate when rendering. Dynamic branching occurs in the shaders to some extent. However, the tests made for the particle system uses a single particle system with a single emitter. The particle system has a total of 3145728 available particles in the particle buffer. The emitter itself is not using any blend mode, it has collisions and painting enabled with a lifetime of 1 second for each particle. The emitter is a cube shaped emitter which has side lengths of 5 units. The particle system is placed in the middle of the Crytek Sponza scene. The tests measures the total amount of particles emitted during a frame, emission time, updating time and rendering time for an increasing emissions rate of particles per frame.

The memory usage of the paint method is calculated in theory by taking the byte size of the texture format and multiply it by the texture resolution, for each allocated texture. The same reasoning can be applied to the particle buffer, but instead taking the size of the particle struct and multiply it by the size of the buffer.
The statistics are gathered in a GPU debugging tool called RenderDoc, which is able to measure timings for each call to the graphics API. No guarantees can be made that RenderDoc provides reliable measurements, however the absolute timings are not of great importance but rather how well the method is able to scale relative to an increasing number of particles or painted meshes. The machine used for testing uses an Intel Core i7-4770K 3.5GHz CPU, 8 GB of DDR3 1600MHz RAM and a NVidia GeForce GTX 760 with 2GB of GDDR5 memory running on Windows 10. As this is the only machine available for the conducted tests there may be problems in replication when using other configurations e.g graphics drivers and graphics card vendors, however no vendor specific technologies was used in the implementation.

4.1 Painting

Time complexity of the paint application method can be seen in table 4.1, and screenshots during the tests in images can be seen in figures 4.1, 4.2 and 4.3. The table 4.1 shows that the time of painting a single cube is $4\mu$s and painting a hundred cubes takes approximately hundred times as long, which gives a time complexity of $O(n)$ where $n$ is the number of cubes being completely covered in paint each frame.

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{fig4.1.png}
\caption{The particle system bounding box covering 1x1 cubes.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{fig4.2.png}
\caption{The particle system bounding box covering 6x6 cubes.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{fig4.3.png}
\caption{The particle system bounding box covering 10x10 cubes.}
\end{figure}

4.2 GPU particle system

The timings of a GPU particle system placed in the middle of the Crytek Sponza scene, rendering opaque particles only. The collected data can be seen in table 4.2 and the images 4.4, 4.5 and 4.6 shows 3 out of 4 test runs with increasing emission rate of particles. All of the emitted particles are allowed to paint the geometry of the scene with a black color when a collision occurs.
Table 4.1: Table of samples taken during the testing of time complexity. The texture format RGBA16F has 4 color channels of 16 bits per channel. Texture resolution is in pixels.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Total time (µs)</th>
<th>Painting time (µs)</th>
<th>Texture resolution</th>
<th>Texture Format</th>
<th>Cubes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1909.76</td>
<td>4.00</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1831.64</td>
<td>16.13</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1891.84</td>
<td>39.49</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>1953.09</td>
<td>83.04</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>2212.94</td>
<td>106.56</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>1993.65</td>
<td>150.59</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>1999.26</td>
<td>196.29</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td>2091.33</td>
<td>257.57</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>2153.89</td>
<td>323.97</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>81</td>
</tr>
<tr>
<td>10</td>
<td>2265.66</td>
<td>415.04</td>
<td>512x512</td>
<td>RGBA16F</td>
<td>100</td>
</tr>
</tbody>
</table>

4.3 Real-time mesh painting with GPU particle systems

The Crytek Sponza scene as well as the Stanford Chinese dragon was used to put the paint application method into a use case which is more similar to those of computer games. The first example of the after effects of painting can be seen in figures 4.7, 4.8, 4.9, 4.10 and 4.11 where a particle system simulating blood splatter is shown. The blood splatter particle system consist of a maximum of 1048576 particles in the particle buffer with a single emitter and a multiplicative blend mode. The emitter has an emission rate of 20 particles per frame and uses a custom texture.

The figures 4.12, 4.13, 4.14 show two particle systems simulating a fire, the fire is moving along a spline on the floor of the Crytek Sponza floor scorching it as particles collide with the floor. The fire particle systems each consist of a maximum of 40000 particles in their respective particle buffers and uses an additive
Table 4.2: Data collected of 4 test runs of the same particle system, using a different emission rate of particles for each test run.

<table>
<thead>
<tr>
<th>Total particles</th>
<th>Emission rate</th>
<th>Emission time ($\mu$s)</th>
<th>Update time ($\mu$s)</th>
<th>Render time ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>53000</td>
<td>1000</td>
<td>31.17</td>
<td>2430.53</td>
<td>319.49</td>
</tr>
<tr>
<td>414720</td>
<td>10000</td>
<td>62.56</td>
<td>3591.49</td>
<td>2443.87</td>
</tr>
<tr>
<td>2284544</td>
<td>100000</td>
<td>569.86</td>
<td>8259.94</td>
<td>16572.74</td>
</tr>
<tr>
<td>3145728</td>
<td>1000000</td>
<td>41.77</td>
<td>8778.46</td>
<td>26367.10</td>
</tr>
</tbody>
</table>

blend mode. The fire particle system has three emitters, one for smoke, fire and sparks. The smoke emitter has an emission rate of 1 particle per frame, the fire emitter has an emission rate of 25 particles per frame and lastly the spark emitter has an emission rate of 3 particles per frame. All of these emitters use different custom textures.

Figure 4.7: Painting applied to the curtains of the Crytek Sponza scene, shows paint applied as dripping due to particles being propagated along the surface of the mesh after a collision.
4.3 Real-time mesh painting with GPU particle systems

Figure 4.8: Painting applied to the vegetation of the Crytek Sponza scene, shows paint can be applied to very small complex meshes.

Figure 4.9: Painting applied to the Stanford Chinese dragon placed in the Crytek Sponza scene, shows paint can be applied to meshes with a high number of vertices.
**Figure 4.10:** Painting applied to the side of a wall in the Crytek Sponza scene, showing that the UV to triangle ratio affects the results. In this case the paint is stretched.

**Figure 4.11:** Painting applied to a hanging cloth in the Crytek Sponza scene, showing that the UV mapping affects the results, in this case overlapping UV coordinates.
4.3 Real-time mesh painting with GPU particle systems

Figure 4.12: Fire particle system moving along spline in the Crytek Sponza scene. The scene took 9.38 ms to render in total, the particle simulation took 1.7 ms and painting took 0.92 ms.

Figure 4.13: Fire particle system moving along spline in the Crytek Sponza scene. The skip in the paint stroke is due to a frame capture which made the program skip a few frames. The scene took 10.14 ms to render in total, the particle simulation took 0.6 ms and painting took 0.91 ms.
Figure 4.14: Fire particle system moving along spline in the Crytek Sponza scene. The paint is reapplied each time the particle system moves over the surface. The scene took 7.95 ms to render in total, the particle simulation took 1.87 ms and painting took 0.91 ms.
In this chapter the visuals and performance of the results are discussed as well as the choice of method. Reaching the goal of the thesis required the implementation of quite a complex system, as such there are many things that need to be taken into consideration. The most important experiences from using and implementing the system in practice is gathered and discussed.

5.1 Results

The results show that the method works in an environment similar to those found in 3D games, where a scene has a large number of meshes that could be subject to painting, as shown in the tests made with the Crytek Sponza scene. The paint application itself has a good enough time complexity to be considered usable in real-time graphics applications, as shown in the cube test. However, with larger texture resolutions of the paint textures, the time of painting would be significantly increased due to more render operations. The resolution of 512x512 pixels for paint textures was enough to mitigate the effects of texture magnification when blended with some octaves of Perlin noise in the tests. The bicubic filtering and noise applied when sampling the paint texture in the deferred geometry pass is not the cheapest, it is certainly a lot more costly than a single texture lookup using bilinear filtering. However, the way the paint texture is used in the renderer is up to the implementer of the paint method, if there’s no requirement for high detail, a single texture lookup may be enough.

How the paint texture looks when mapped to the surface of a mesh is entirely up to the UV mapping, a mesh occupying a small screen area would be able to maintain detail with smaller paint texture resolutions. One way to maintain detail is therefore to reduce the screen area occupation of triangles, i.e, the large floor plane of the Crytek Sponza scene could be split into separate meshes, each
with their own paint textures. Because the paint method relies completely on the
UV mapping of the models some consequences of this can be seen in figures 4.10
and 4.11 where the first one has a bad triangle to UV ratio because the original
model was made to have a repeating texture, the second one has triangles with
overlapping UV coordinates making the paint application mirror itself. The solu-
tion to this would be for artists to simply know what objects that could be subject
to painting and separate meshes so that UV coordinates and triangle ratios would
work well together with the painting, this would of course introduce additional
draw calls due to more meshes being rendered, but would also imply that more
meshes could be culled away with view frustum culling.

The rendering times for the frames shown in figures 4.12, 4.13 and 4.14
slightly change because of the view frustum being oriented differently and the
view frustum culling. The rendering time of the particle systems change some-
what because they are subject to quite a lot of overdraw when rendering, as the
particle systems move further away from the observer the area of overdraw is re-
duced and there is a performance increase. The timing of the paint applications
stays almost the same for each frame, that is because the paint application only is
affected by the texture resolution and the amount of vertices in the painted mesh.
Because a constant texture resolution was used for each mesh in the scene and
the meshes have a similar amount of vertices the same timings can be expected.
The only exception is when the particle systems intersect the Stanford Chinese
dragon which has a lot of vertices, that increases the painting time slightly.

The tests done for the particle system shown in figures 4.4, 4.5 and 4.6 and
in table 4.2 shows that there are many variables when rendering a particle sys-
tem and it is hard to get consistent and reliable results. It is shown that the the
lifetime of a single particle in conjunction with the emission rate of a particle
emitter has a large impact on the amount of rendered particles that are active
per frame. When the density of particles increases, the time it takes render the
particles increases to exceed the time it takes to simulate them due to overdraw.
An anomaly in the test results was also found when having an emission rate of
100000 particles per frame where the emission time got 10 times that of the other
tests, we can only speculate why this happens. Perhaps the anomaly is due to poor
cache behaviour if the index values of the particle buffer found in the dead list
are fragmented, or perhaps it is due to some worst case scenario of the dynamic
branching done in the emitter when comparing the dead count and the thread
ID.

Allocating any more than 3145728 particles in the particle buffer on the test
machine would cause the particles to simply not show when running the program,
which is probably a limitation of either the graphics driver or the VRAM. In the
case where the emission rate is 1 million particles per frame, the amount of active
particles during the frame is the total of particles in the particle buffer. For this
particular particle system, this is the worst possible case, we have many particles
in the same area which causes a lot of overdraw which in turn affects the fillrate
of the GPU. Having a larger particle size with alpha-blending such as smoke or
similar would be even worse in terms of the amount of overdraw.
5.2 Method

The method was required to work on Direct3D 11 with a deferred renderer, however the method showed to be independent of the real-time renderer type. A forward renderer could implement the method in a similar manner as the deferred renderer implementation. Because the scene geometry is already rendered into the frame buffer before rendering particles, the scene depth buffer is readily available. The scene normals provided by the G-buffer in a deferred renderer could be computed by taking the forward, backward or central difference between depth samples or by using a thin G-buffer with normals and depth written during a pre-pass, which is sort of a hybrid approach.

The painting and the particle system is separable, where one does not require the other. That means that existing games could probably implement the painting quite easily by adding a screen space particle collision texture as an RTV to an existing GPU particle system framework. The paint application is fast because it requires only a single texture lookup to determine if there was a particle collision at a specific screen space position. This method compared to the only previous work found by Drone [4] does not require any iteration over the particle collisions in the pixel shader, which becomes costly if there are many particle collisions.

Painting on meshes with a higher vertex count increases the time slightly due to these transformations, however the increase in time is not something that caused a problem during testing. The painting method does not require shader model 5.0 but the GPU particle system does, due to the extensive use of UAV resources. The pre-allocation of textures was not the initial thought, but rather a fix to framerates dropping too low when allocating them dynamically as particle systems intersected a mesh, the pre-allocation provides more control and do not have the same hiccups. There are some things I would have changed, but it has not got much to do with the actual painting method or GPU particle system, but rather the rendering framework that this whole project is built on top of. It took too much time out of the actual focus of the thesis work, and I would have liked to experiment more with the actual paint method, how different types of material properties could be changed rather than just the diffuse color of the meshes.

There are some interesting things that could possibly be implementing, such as accumulating puddles of water due to rain by changing diffuse color and specularity, or a build up of snow using diffuse color and using the paint texture as a dynamic height map or displacement map. Another interesting area of research is how the paint texture could be used as the weight texture for texture splatting, perhaps a small texture could be repeated in areas of the weight texture using texture bombing. The thing that would improve the particle system the most is if the render list was sorted so that alpha-blended particles are rendered in the correct order. In the case of the painting method, it would be interesting trying to create some parametrization of the surface of a mesh such that smaller textures could be used for parts of the mesh. Being able to map textures to areas of the mesh instead of the whole mesh could improve the method by using less memory and give better detail.
The purpose of this thesis work was to find a method for painting meshes in a 3D game scene with the help of particles simulated on the GPU. The argument was that such an effect would be able to add detail and have particle collisions leave a lasting impression on the scene. The real-time mesh painting was achieved by using screen space methods to determine intersections between scene meshes and particles, painting the mesh at the intersection points. The results of the painting is output to a texture which is fed back into the renderer when rendering the mesh. The paint method is fast and has a linear time complexity which enables it to scale well as GPUs get more powerful.

Meshes can be painted at real-time frame rates, where the number of meshes that can be painted in one frame depends on the complexity of the meshes. The reason for this is that the method requires a separate geometry pass for painted meshes. The performance and precision of the method also depends on the resolutions used for the paint textures. Details in the paint textures can be retained and added by using bicubic filtering and noise functions, which removes the smooth look of bilinear filtering, giving a sharper paint application. Since the paint method is dependent only on the previously mentioned factors as well as the UV-mapping of the painted mesh itself it can be concluded that the amount of simulated particles does not affect the time of painting a mesh directly.

The drawbacks of the method can be seen when: a triangle of the mesh is large in screen space but using a small area of its UV-space, causing stretching of the paint; a triangle of the mesh overlaps in the UV-space, causing mirroring of the paint. By pre-allocating paint textures and dynamically assigning them to meshes intersecting the particle systems, the method can be used in scenes with a large amount of meshes, similar to those found in 3D games.

The found method is independent on the type of renderer used and can be implemented by adding on to existing GPU particle system implementations.
Bibliography


