GParticles: a flexible GPU-based particle library

Tiago Dinis
Departamento de Informática
Universidade do Minho
Email: tiagoddinis@gmail.com

António Ramires Fernandes
Centro Algoritmi
Universidade do Minho
Email: arf@di.uminho.pt

***Abstract*—Particle systems are widely used to create visual effects with a large number of elements (or particles). Earlier works were CPU based, while recent developments use the GPU to take advantage of the fact that, in most cases, particles are independent entities. In here, we propose a GPU-based library that focuses on extensibility and ease of use. The library allows for virtually any particle logic to be implemented, supporting fully programmable shader based effects, facilitating the definition of multiple resources and providing full control over the particle system's simulation execution. Projects can be easily specified using XML, and extended via shaders and stubs, C functions that can be executed in between particle system's processing stages. The examples presented illustrate the potential of the API, with multiple particle systems acting in a sequence, later extended to create custom graphic effects.**

***Index Terms*—particle systems, gpu simulation, real time graphics.**

I. Introduction

A particle system is a collection of individual entities that have both a set of data and behavior rules. During their life cycle, these entities, or particles, can interact with the environment and with each other, forming a large-scale system that models a visual effect. Particle systems are extensively used in computer graphics to reproduce fuzzy and natural phenomena, volumetric effects or large, behaviorally complex systems, e.g, simulations of birds flocking, a fire and its cinders, or imaginary fluids with unrealistic dynamics.

Some particle systems are CPU-focused [7][9]. On these implementations, the entirety of the system data is managed on the CPU and only transferred to the graphics hardware for the rendering stage. Due to the CPU's higher flexibility over the GPU, particle systems that use this approach are easier to program and design [10]. However, the transfer bandwidth of particle data for rendering purposes can be a major bottleneck and severely limit the maximum number of particles rendered each frame.

One viable alternative is to delegate particle system simulation stages to the GPU [8][11][11][13][14][5]. As a result, particle data communication is reduced, allowing the development of systems composed by millions of particles, and the use of previously owned CPU processing power for other application tasks. This approach will also leverage most particle system simulations, since they can be modeled as a highly parallel workload that fits well with the GPU.

In this paper we present a GPU-centric particle library[1] whose architecture aims at extensibility and user-friendliness. The system is capable of high performance since particle data is GPU resident, therefore limiting/eliminating data transfer bottlenecks. The library remains flexible and supports particle systems with virtually any behavior without the need to be recompiled every time something is changed. It provides a collection of powerful and convenient utilities to create and extend particle systems, defined through an XML file and custom functions that act as replacements for pre-existing code. Furthermore, the library allows full control of the execution of multiple particle systems at a fine grain, allowing, for instance, the sequencing of effects or the definition of triggers to change/start/stop a particular particle system.

The remainder of this document is structured as follows: section II presents some of the related work, focusing on GPU-centric systems; section III describes the architecture of the proposed library; section IV goes through the available extensibility options for particle system development; section V details GParticles' important utility features; section VI provides a simple example showcasing common features of GParticles; section VII presents a more complex example that models, with GParticles, a crude animation system where each object is viewed as a particle; section VIII provides a performance evaluation report; section IX concludes the paper and presents avenues for future work.

II. Related Work

Particle simulation can be done through stateless or stateful systems.

Stateless (or parametric) systems, do not preserve state between iterations [2]. Instead, they use finite mathematical expressions with a set of initial values and global parameters, such as elapsed time, gravity force, and starting particle positions and velocities, to compute a particle's current properties [1]. This approach results in efficient simulations and is easily implemented on the GPU. However, since particle behavior is attached to a deterministic expression, it soon becomes unfeasible to model interactions with the environment and differing simulation paths for individual particles. As a result, the stateless approach is limited to simple, isolated particle systems.

[1] https://github.com/tiagoddinis/GParticles

Stateful (or non-parametric) systems store particle data throughout iterations, applying numerical integration methods to calculate the current state of the system. While harder to use on the GPU, stateful systems are better suited for environment interaction simulations and branching particle behavior, e.g., collisions, randomness, agent intelligence.

Transform feedback was added to the OpenGL version 3.0 improving the implementation flexibility of GPU stateful particle systems. With this technique, a buffer is connected after any vertex processing stage (vertex, geometry or tessellation shaders) to capture generated/transformed data. The transform feedback buffer can then be used to provide the output vertices of the current pass as input for the next draw call. Transform feedback works as an extension of the traditional GPU pipeline and some examples are available. [3][5][6].

Prior to the existence of transform feedback, GPU-based particle systems were usually implemented with fragment shaders, FBOs and textures, where each texture (or each texture row) could represent a particle property, for instance, particle position and velocity [1][5]. Floating-point textures were also made core in version 3.0 and, with that precision increase, GPU particle systems started being taken more into consideration. This approach was preferred for a time, having been used in some commercial products. An example is Kvant/Spray [8], described as "a GPU accelerated object instancing/particle animation system for Unity", which uses floating-point textures to store system state. ParticlesGPU is another engine that uses the same method [13]. It was developed on top of vvvv [12], a multipurpose toolkit (free for non-commercial use) suited for large media environments with physical interfaces and real-time motion graphics. Unfortunately, these examples are both lacking in documentation and confined to the engine/toolkit they were built upon. ShaderParticleEngine [11] is a library focused on GPU particle systems for web browsers. Using this API as is provides a solution for simple particle systems.

Compute shaders, introduced in OpenGL 4.3 core version, work as single-stage programs (completely separate from the traditional GPU pipeline) used to compute arbitrary data. They work similarly to other shader stages, but the work load, set of input and output values, is not predefined. Compute shaders are a perfect fit for many tasks, including the processing of particle data.

In the 2014 edition of the Game Developers Conference (GDC), Gareth Thomas' presentation [15] gave an overview of a compute-based particle system architecture and how some common functionality, such as collisions, sorting and rendering could be achieved in this new context. The Doom video game released in 2016, praised for its stunning visuals, already presented a graphic option to enable GPU particles, simulated through compute shaders [14].

While there are many demos available of particle systems using compute shaders for particle physics, our research was unable to find an open-source library focused on the development of GPU particle systems that allowed users to easily extend effects at different levels of abstraction, from high-level constructs down to the shaders themselves, and that could easily be integrated with any C++ application.

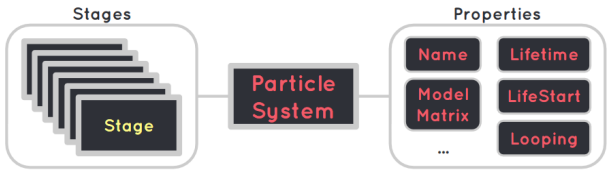

Fig. 1. GParticles particle system structure

III. Architecture

This section gives an overview of the various elements that compose GParticles: subsection III-A explores how data is organized, stored and accessed; III-B presents particle system's and stage's architecture; III-C introduces GParticles main high-level components; III-D shows how a GParticles iteration is processed.

A. Data in GParticles

As previously stated, a particle system is characterized by having a large collection of data. GParticles allows a system to have its relevant information spread across several data resources, which can be one of three OpenGL shader storage constructs: buffers, atomic counters (or atomics) and uniforms. These data resources have distinct behaviors and use cases, and are either global in scope or relative to a single particle system.

Buffers are fixed size arrays with elements of a certain type. They are the go-to resource to represent properties of particle instances, such as their lifetime or position. Buffers can also be used to communicate between different particle systems.

Atomics are unsigned integer hardware counters that can only be incremented, decremented, and have its value read. Although their functionality is limited, atomics are very useful in such a parallel environment as the GPU. Atomics can be used to keep track of buffer indices, elapsed processing iterations or the number of alive particles.

Uniforms work as shader program inputs from the application side, being immutable in value during the execution of the program. Global variables and state such as the elapsed time since the last frame, or mouse and keyboard state, can easily be passed on to the GPU through uniforms.

B. Particle Systems and Stages

A particle system has a set of properties and can have an arbitrary number of stages, see Fig. 1.

Properties are general pieces of information that describe the system, such as its name, model matrix, control flow variables, or the number of work groups to be used in computation stages.

Stages are objects that refer to a shader program, either made of a single compute shader, or vertex (including vertex, tessellation and/or geometry) and fragment processing shaders. Stages are tasked with processing particle data with a specific goal. Most particle systems use three stages:

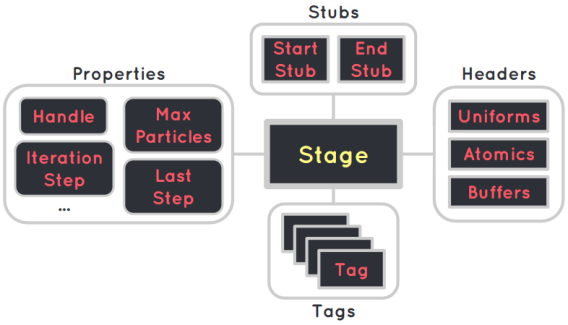

Fig. 2. GParticles stage structure

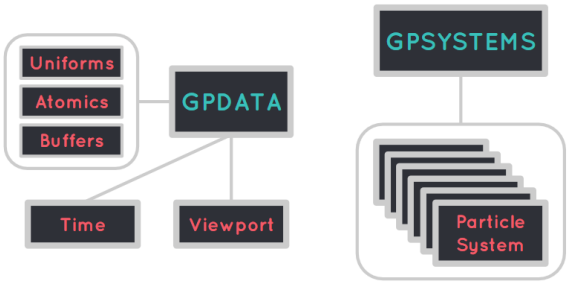

Fig. 3. *GPSYSTEMS* and *GPDATA* components

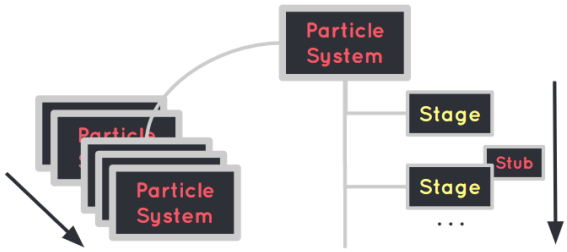

Fig. 4. GParticles iteration example

- **Emission:** controls how new particles are instantiated and initializes their properties;
- **Update:** alters particle data according to the specified behavior. It's also the stage that tests for collisions and decides when particles should die;
- **Render:** defines the graphical representation for each particle;

To increase flexibility, GParticles allows a particle system to have an arbitrary number of stages.

Stages are defined by their properties, headers, tags and stubs, as seen in Fig. 2. Examples of stage properties are: the shader program handle, maximum number of particles, and control flow variables, such as the iteration step or last iteration timestamp. The header set relates the resource name and the corresponding shader binding point.

Tags are strings set by the user that provide additional information about the system. Some tags such as "emission", "update" and "render" indicate GParticles how the stage program should be generated. Custom stages can be tagged as "compute" and "graphics", generating a single compute program stage or one that uses the traditional graphics pipeline, respectively. The "paused" and "active" tags signal GParticles if the stage is to be be executed. The user can also add tags that have no particular meaning to GParticles, but give useful clues for operations he implements, as will be shown in section VII.

Finally, each stage can receive user-defined C/C++ functions named stubs that control, from the CPU side of the application, particle system behavior at runtime. Stubs fill one of two stub stage slots: "startStub" and "endStub" being called before or after the execution of the stage shader program, respectively.

C. GParticles main components

GParticles strives to be simple in design, with only three main architectural components: two singletons, *GPDATA* and *GPSYSTEMS* (shown in Fig. 3), and the project loader.

GParticles data singleton, *GPDATA*, is responsible for storing all the available data resources. Through *GPDATA* it's possible to access, update, add or remove resources; it also provides some other useful data, such as the viewport dimensions or the current time in various time units.

GPSYSTEMS holds the loaded particle systems and controls their execution flow. With this singleton, it's possible to access, update, add, or remove particle systems and its stages. It's also through *GPSYSTEMS* that one can select particle systems to iterate their simulations.

The project loader loads an XML file with the particle systems configuration, processes that information and fills the two singletons with the required data to run the simulations.

D. GParticles iterations

Once a project definition file has been parsed by GParticles' loader, the GPSYSTEMS singleton `processParticles` function can be called, accepting model, view and projection matrices as parameters (which default to identity matrices if none is provided) and a set of tags. When called, `processParticles` iterates through particle systems that have stages with the corresponding tags and invokes their `execute` function, which will begin to process its active stages. If no tag is selected, every particle system will be simulated.

When its time for the execution of an active stage, the concrete stage class binds the required resources through *GPDATA*, loads the corresponding program object and runs it. If present, stage stubs are executed before and/or after the program object. A GParticles iteration is represented in Fig. 4.

IV. GParticles Extensibility

GParticles allows users to design and extend particle systems at different abstraction levels.

The first level is the XML definition file, where it's possible to alter project configurations, resource values and, through prefabs, quickly exchange between sets of particle system data and behavior logic. This level of control, however, is dependent on previous implementations; it limits itself to changes

```
void main() {
    // exit shader if particle is dead
    if (@lifetimes[gid] == -1) return;

    // age particle
    @lifetimes[gid] -= @deltaTime;

    // if particle just died, set its lifetime to -1,
    // decrement number of alive particles and exit
    if (@lifetimes[gid] <= 0) {
        @lifetimes[gid] = -1;
        atomicCounterDecrement(@aliveParticles);
        return;
    }

    // execute custom logic of the update function
    update();
}
```

Fig. 5. GParticles template updateMain source

of pre-defined particle properties (their color or lifetime, for instance) and to the choice of logic presets, such as switching a system's emission primitive from cone to sphere. Nevertheless, this level of control is useful for fast prototyping, quickly assembling a base configuration and tweaking implementation exposed variables.

The next level of extensibility are the shader files of each stage. These are composed by templates, custom logic functions and modules. Template files contain the `main` shader function and generic functions that work as a cohesive logic construct, common to most particle systems. Templates also call one or more custom functions. These functions are usually created on a separate file and define the custom operations to be applied on particle data during the execution of the corresponding stage. Currently, GParticles has four different templates:

- **emissionMain:** calls the custom `emission` function if the maximum number of alive particles the system allows and the number of particles to be emitted per iteration limits haven't been reached;
- **updateMain:** ages particles and calls the custom `update` function if they are still alive, see Fig. 5;
- **updateCollisionMain:** works the exact same way as updateMain but, if the particle is currently alive, another custom function, `collision`, is called after `update`, testing particles against collider primitives defined on the XML project file;
- **billboardMain:** calls the custom `transform` function that must return an array with four vec4 elements representing the billboard quad coordinates. These coordinates are then offsetted by the particle origin position and projected to screen space. The quad texture coordinates are also set;

Modules are files that provide a set of auxiliary functions related to specific tasks. Module functions are mostly independent between them and can be used by custom logic files and templates alike. At the writing of this paper, there are

```
GP_Uniform u;
if (GPDATA.getUniform("myUniform", u)) {
    GPDATA.setUniformValue("myUniform", u.value() + 1);
}
```

Fig. 6. Incrementing a uniform resource value

three available modules: "utilities" provides general helpful functionality, such as random number generation, construction of rotation matrices and 3D spaces; "emission" facilitates initialization of data relative to 3D particle properties with emission primitives and velocity vector generators; "billboard" has auxiliary functions for billboard manipulation (creation, stretching and orientation).

Templates, modules and custom logic files are only conceptually distinct; they are concatenated into a single shader file. However, this distribution of tasks can be a very useful convention to follow when using GParticles. Users can add new templates, or just redefine a custom function as detailed in section VI. New modules can also be written and imported from the XML project file.

From the CPU application side the user can extend particle system behavior by directly manipulating data resources through *GPDATA*. Fig. 6 demonstrates how a float uniform could be incremented.

Another powerful application-side extensibility option are stubs. Having full access to the GParticles API, stubs can implement virtually any additional logic required for a given effect. They can do things such as changing data resources values and particle system execution flow in runtime, or post-stage processing operations. Section VI showcases an example where stubs play an important role.

V. Utility Features

This section details some features of GParticles that facilitate the development of projects both for XML and shader programming.

A. Header Generation

With the loading of a project, GParticles generates automatically all the required shader headers, the portions of code that declare the available inputs and variables. This process is based on the resource configurations indicated on project files.

Listing in Fig. 7 presents an example for a compute shader.

B. The prefab tag

GParticles allows the use of a prefab tag to facilitate the creation of project definition files, their readability, and reuse of configurations. A prefab tag references a previously defined XML file, inserting its content at the tag location during the project loading process. Prefab files can have slot tags which signal entry points for additional configurations (defined on the project file inside inject tags). Injected tags override prefab tags with the same name attribute. Fig. 8 shows how GParticles creates the final file to be loaded, from the original XML project and a prefab.

```
>> project.xml

<project>
  <resources>
    <instance>
      <buffer name="positions" elements=512 type="vec4" />
        ...
      <atomic name="aliveParticles" />
        ...
      <uniform name="maxParticles" type="float" value=10 />
    </instance>
  </resources>

  <psystem name="ps">
    ...
-----------------------------------------------------------------
layout(binding = 0, offset = 0)
  uniform atomic_uint ps_aliveParticles;
    ...
layout(std430, binding = 9) buffer Ps_positions
  { vec4 ps_positions[]; };
    ...
uniform mat4 model, view, projection;
uniform float ps_maxParticles;
```

Fig. 7. Shader header generation from project file configurations

```
>> project.xml
<project>
  <prefab path=".../defaultResources.xml">
    <inject slot="instance">
      <buffer name="newProp" elements=512 type="vec4" />
      <uniform name="maxParticles" type="float" value=10 />
    </inject>
  </prefab>
  ...
</project>
-----------------------------------------------------------------
>> defaultResources.xml
<resources>
  <instance>
    <slot name="instance" />

      <buffer name="lifetimes" elements=512 type="float" />
      <uniform name="maxParticles" type="float" value=512 />
    ...
</resources>
-----------------------------------------------------------------
>> generated_project.xml
<project>
  <resources>
    <instance>
      <buffer name="newProp" elements=512 type="vec4" />
      <uniform name="maxParticles" type="float" value=10 />
      <buffer name="lifetimes" elements=512 type="float" />
      ...
</project>
```

Fig. 8. Exemplification of the use of a prefab

```
<project>
  <prefab path=".../defaultResources.xml" />

  <psystem name="mypsystem">
    <properties>
      <position x=0 y=0 z=-5 />
    </properties>

    <stages>
      <prefab path=".../defaultEmission.xml" />

      <prefab path=".../defaultUpdate.xml" />

      <prefab path=".../pointRender.xml" />
    </stages>
  </psystem>
</project>
```

Fig. 9. A simple particle system using prefabs

C. The '@' directive

GParticles provides the '@' directive as an easy, generic way to reference instance resources. During the project loading phase, whenever '@' is found on a stage file, it is replaced by the name of the particle system that particular file is being added to. For instance, writing the instruction `@positions[gid].y -= @deltaTime` inside a stage file referenced by a particle system named "rocket" would be the same as writing `rocket_positions[gid].y -= rocket_deltaTime`. Creating stage files with '@' allows for clear identification of particle system variables within the code. Furthermore, it also promotes code reuse, since, if two particle systems have the same logic, there is no need to create a file for each one just to reference the differing instance resources. GParticles' template files are a great example of this, making extensive use of the '@' directive so they can be referenced by different particle systems, as long as these systems provide the required instance resources.

VI. Example project

This section will give a walk-through on how a basic particle system could be extended to create another effect.

A GParticles project starts with the creation of an XML file that defines the desired particle systems, their corresponding properties and stages, and indicates what resources are to be used. This example begins with a particle system positioned 5 units along the negative Z axis, with default stages and resources, referenced through prefabs, as shown in Fig. 9.

Once emitted, particles rendered as points will travel in a straight line, following their position vector direction, and die after a default amount of time.

The goal is to alter the current project so that it resembles a water fountain with particles that bounce from the ground. To achieve this goal, the emission shape is changed from the default sphere to a cone by injecting an overriding tag (that references a new file with the custom emission shader function) in the corresponding prefab `customFunction` slot. The particle rendering stage is also changed from points

```
>> project.xml
    ...
    <stages>
      <prefab path=".../defaultEmission.xml" >
        <inject slot="customFunction">
          <file name="custom"
                path=".../customEmission.glsl" />
        </inject>
      </prefab>

      <prefab path=".../defaultUpdate.xml" />

      <prefab path=".../billboardRender.xml" />
    </stages>
    ...
-------------------------------------------------------------
>> customEmission.glsl

void emission() {
  ...
  //@positions[gid] = emitFromSphere(1.5, false);
  @positions[gid] = emitFromCone(1.5, 3, false, true);
  ...
}
```

Fig. 10. Project definition with a custom emission function

```
void update() {
  // update velocity applying gravity
  @velocities[gid] -= 9.8 * @deltaTime;
  // test particle position against plane Y=-2
  if (@positions[gid].y < -2) {
    // damp velocity in Y coordinate
    @velocities[gid] *= 0.5;
    // set collision to true (initial value is false)
    @bounced[gid] = 1;
  }
  // finally update the positions
  @positions[gid].xyz += @velocities[gid].xyz * @deltaTime;
}
```

Fig. 11. Custom update shader function

to velocity-aware billboards, by selecting another prefab. The relevant changes are presented in Fig. 10.

The next step is to replace the default update shader function in order to add gravity to particle movement, define a bouncing plane at Y = -2, and determine if a particle has bounced from the plane.

To extend the example further, particle aging should start only after the first bounce, which implies that we must record if a particle has bounced. Bounces can be recorded in a new particle property, `bounced`, injected in the resources prefab in the ”instance” slot. The values of `bounced` are set to zero on particle emission and set to one when the first bounce occurs.

A new file named ”customUpdate” is created where, just like in the prefab default update function, the particle’s velocity is added to its position. Prior to that, velocity is updated to reflect the constant gravity force. Afterwards, the particle is tested against the plane. To simulate the bouncing effect, the Y component of velocity is inverted and weighted with a damping coefficient. The shader code is presented on Fig 11.

The bounced trigger can now be used to set in motion the

```
void main() {
  if (@lifetimes[gid] <= 0) return;
  if (@bounced[gid] == 1) {
    // particle aging and death functionality
    ...
  }
  // calls custom update function
  update();
}
```

Fig. 12. Custom update stage adapted from updateMain template

```
<project>
  <prefab path="prefabs/resources/defaultResources.xml">
    <inject slot="instance">
      <buffer name="bounced" elements=512 type="float" />
    </inject>
  </prefab>

  <psystem name="mypsystem">
    <properties>
      <position x=0 y=0 z=-5 />
    </properties>

    <stages>
      <prefab path=".../defaultEmission.xml" >
        <inject slot="customFunction">
          <file name="custom"
                path=".../customEmission.glsl" />
        </inject>
      </prefab>

      <stage>
        <tag name="update"/>
        <file path=".../customMain.glsl" />
      </stage>

      <prefab path=".../billboardRender.xml" />
    </stages>
  </psystem>
</project>
```

Fig. 13. Final XML project definition

aging of particles. This is a rare occasion where the update stage template requires modifications, because its default particle lifetime control does not align with the desired effect. The update template contents are copied to a new file and an `if` condition is added so the particle is aged only if it has already bounced from the floor, see Fig. 12.

Since we’re using a custom update stage we must define it in the XML file. Fig. 13 presents the complete XML project and Fig. 14 shows both the original and new visual effects.

This example covered common GParticles extensibility options. To further extend the effect, the user could add particle systems, stages and the required resources to the project file. Section VII goes more in-depth on the development process of complex projects, composed by multiple particle systems and stubs.

VII. Particle system with animation phases

This section will walk through the necessary steps to implement a more complex project in GParticles. This project

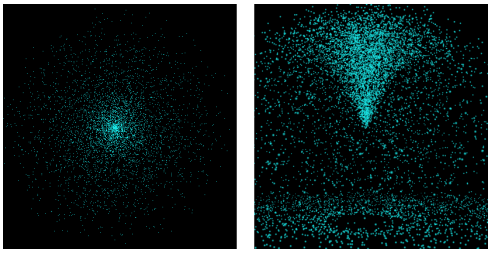

Fig. 14. Default (left) and water fountain (right) particle systems

```
<project>
  <prefab path=".../defaultResources.xml" />

  <psystem name="auxiliary">
    ...
    <override type="uniform" name="maxParticles" value=5 />
  </psystem>

  <psystem name="main">
    ...
    <override type="uniform" name="maxParticles" value=1 />
  </psystem>
</project>
```

Fig. 15. Overriding default properties for different particle systems

```
void emission() {
  @lifetimes[gid] = 3600;
  @colors[gid] = vec4(0.6,0.1,0.1,1);
  @velocities[gid].xyz = vec3(0,0,-2);

  // places auxiliary propellers around
  // the main propeller (that has radius 8)
  float angle = radians(72 * gid);
  @positions[gid] = vec4( cos(angle) * 8,
                          sin(angle) * 8,
                          0, 1);
}
```

Fig. 16. Custom emission function for propellers

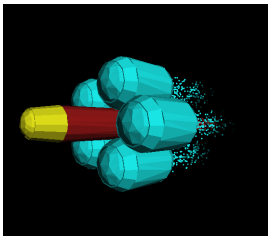

Fig. 17. First shuttle phase

features a space shuttle with two types of propellers, each with a different propulsion trail color. With a key press, the shuttle has its auxiliary propellers fuel depleted, with a visible propulsion trail waning, detaching them as a result. To demonstrate GParticles flexibility and potential, even for simple interactive animation systems, everything in the example will be a particle, including the shuttle itself.

The first step is to create the project definition file and to reference the ”defaultResources” prefab. Then, two particle systems named ”auxiliary” and ”main” are created with a position property where the z coordinate is -50. The ”auxiliary” particle system will override the ”maxParticles” uniform from ”defaultResources” to 5 while ”main” will do the same thing to 1, see Fig. 15.

Next, a stage with the ”emission” tag is added to each particle system. Inside the stage tag, two files are referenced: the custom stage file and the ”emissionMain” template. On the custom stage file the shuttle elements are given a color, a lifetime and a position. The ”main” component sits at the origin of its particle system matrix while the ”auxiliary” components are offsetted around the z axis with the radius of the ”main” component model. The required code for the auxiliary propellers is presented in Fig. 16.

Another stage with the ”render” tag is added to each particle system, referencing inside it the model asset path, the ”utilities” module, one custom vertex and one custom fragment stage files. These custom files draw the model by offsetting its vertex by the particle position and follow the Phong shading model, see Fig. 17.

At this point, if the project is run, the shuttle components are visible allowing their positioning, rotation and scale to be easily tweaked. This process should be done with the addition of uniforms to the project file instead of hard coding values. After every shuttle component is set on the desired position, the propeller trail particle systems are introduced following the same process, but with more instances allowed per system and a point rendertype. These trails should be animated in time and so, a stage with the ”update” tag is added. That stage references the ”updateMain” template and a custom file where the particle systems are incremented with their velocity vector, which is set on the emission stage to -10 on the z component.

The auxiliary propellers detachment behavior is the next step (see Fig. 18). On the emission stage, the ”auxiliary” system propeller particles velocity property becomes equal to their position offset with the z component inverted, so the propeller seems to detach itself on the opposite direction of the moving shuttle. The update stage of the trail particle systems can be reused to apply the detachment action.

However, instead of being triggered, the propeller detachment happens immediately when the project is run. The last step is to add the ”paused” tag to the update stage and to create a stub from the application side to remove that tag when the spacebar key is pressed, effectively triggering the detachment, see Fig. 19.

Another stub could be created to extend this behavior

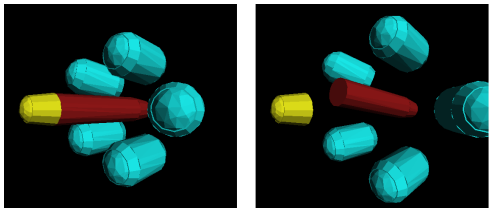

Fig. 18. Shuttle detachment phases

```
// lambda function allows stub to access its stage directly
auto stateToDetach = [&](AbstractStage *as) {

  // check if spacebar is being held, continues if it is
  GP_Uniform spaceHold;
  GPDATA.getUniform("spaceHold", spaceHold);
  if (spaceHold.value() == 0) return;

  // sets state uniform to 2 (propellers detachment)
  GPDATA.setUniformValue("state", 2);

  // triggers auxiliary propeller detachment
  // by removing its update stage tag "paused"
  AbstractStage *stg = GPSYSTEMS.getPSystemStage(
                    "auxiliary", { "update", "paused" });
  stg->removeTag("paused");

  // stub removes itself from the stage slot
  as->setStub("startStub",
    std::function<void(AbstractStage *)>(nullptr));
}
```

Fig. 19. Stub that triggers detachment phase

POINTS	100	10000	1M
Default	158.84 fps	157.72 fps	111.51 fps
Fountain	159.89 fps	158.84 fps	85.41 fps

BILLBOARDS	100	10000	1M
Default	163.19 fps	162.28 fps	62.66 fps
Fountain	160.30 fps	153.92 fps	59.67 fps

CUBES	100	10000	1M
OpenGL test	250.96 fps	247.24 fps	10.44 fps
Fountain	158.37 fps	149.25 fps	3.4 fps

Fig. 20. GParticles performance tests results

by first making the trail particles loose speed and fade, changing uniforms that control those properties. More shuttle components and corresponding detachment phases could be added following the same thought pattern: particle systems with a halted update stage that is activated through stubs and a triggering input. Since adding these stages would lead to the repetition of many logic elements, the use of prefabs should be considered, as it would save development time and improve the project readability as it grows.

VIII. Performance Report

Although the focus has been primarily on the development of a generic architecture for particle simulation, its performance is nevertheless relevant. The fps of the particle systems depicted in Fig. 14 was captured with differing entity numbers (100, 10000, 1 million) and rendering modes (point, billboard and model). The testing was performed with an NVIDIA Geforce 630M (2GB) (see Fig. 20). The entry labeled OpenGL test draws the same amount of geometry as the particle system but at random positions. This enables us to have a glimpse of the relative performance of GParticles. Clearly there is still room for improvement, for instance exploring the workload distribution in the compute shaders, nevertheless, these initial results seem encouraging.

IX. Conclusion

A flexible and fully extensible GPU-centric particle system library was proposed. For the simplest cases, a particle system can be created resorting only to an XML file configuration that references behaviour defined in the library's default prefabs. Systems with more complexity may require the definition of shader code, either the full template or just the custom functions, to provide the additional functionality not present in default resources, such as a new process to define the properties of emitted particles, or how thei are updated.

Further control over particle systems can be obtained by programming with the CPU API, from the flow of particle system execution to the values of data resources. Stub functions allow the definition of all kinds of behaviors to be executed before or after stages.

It was also shown that the library potential goes beyond traditional particle systems, particularly with the simple animation system example.

As future developments we would like to perform an in-depth study on the performance of GParticles, and add a drag and drop editor that would aid the creation of the XML project files.

Acknowledgment

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

References

[1] L. Latta, *Building a Million-Particle System* Internet: http://www.gamasutra.com/view/feature/130535/building_a_millionparticle_system.php, Jul. 28, 2004.

[2] D. Walton, *Stateless Particle Systems* Internet: http://drwalton.github.io/2013/11/06/stateless-particles.html, Nov. 06, 2013.

[3] E. Meiri, *Particle System using Transform Feedback* Internet, http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html, Aug. 28, 2011.

[4] OpenGL.org, *Transform Feedback* Internet: https://www.opengl.org/wiki/Transform_Feedback, Aug. 17, 2016.

[5] P. Rideout, *Noise-Based Particles, Part II* Internet: http://prideout.net/blog/?tag=opengl-transform-feedback, Jan. 29, 2011.

[6] M. Bubnar, *Particle System tutorial* Internet: http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=26, Jul. 15, 2013.

[7] Unity Technologies, *Unity's particle system manual* Internet: https://docs.unity3d.com/Manual/class-ParticleSystem.html, 2016.

[8] K. Takahashi, *Kvant Spray* Internet: https://github.com/keijiro/KvantSpray, Jun. 26, 2016.

[9] Epic Games, *UE4 Cascade Particle Editor* Internet: https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/Cascade/, 2016.

[10] Epic Games, *Unreal Engine 4 CPU vs GPUSprite comparison* Internet: https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/EffectsGallery/1_A/, 2016.

[11] L. Moody, *ShaderParticleEngine* Internet: https://github.com/squarefeet/ShaderParticleEngine, Apr. 08, 2016.

[12] vvvv, *vvvv* Internet: https://vvvv.org/, Jul. 17, 2014.

[13] dottore, *ParticlesGPU* Internet: https://vvvv.org/contribution/particlesgpu-shader-library, Nov. 09, 2010.

[14] A. Courrèges, *Doom graphics study* Internet: http://www.adriancourreges.com/blog/2016/09/09/doom-2016-graphics-study/, Sep. 09, 2016.

[15] G. Thomas, *Compute-Based GPU Particle Systems* Internet: http://twvideo01.ubm-us.net/o1/vault/GDC2014/Presentations/Gareth_Thomas_Compute-based_GPU_Particle.pdf, Mar. 17-21, 2014.