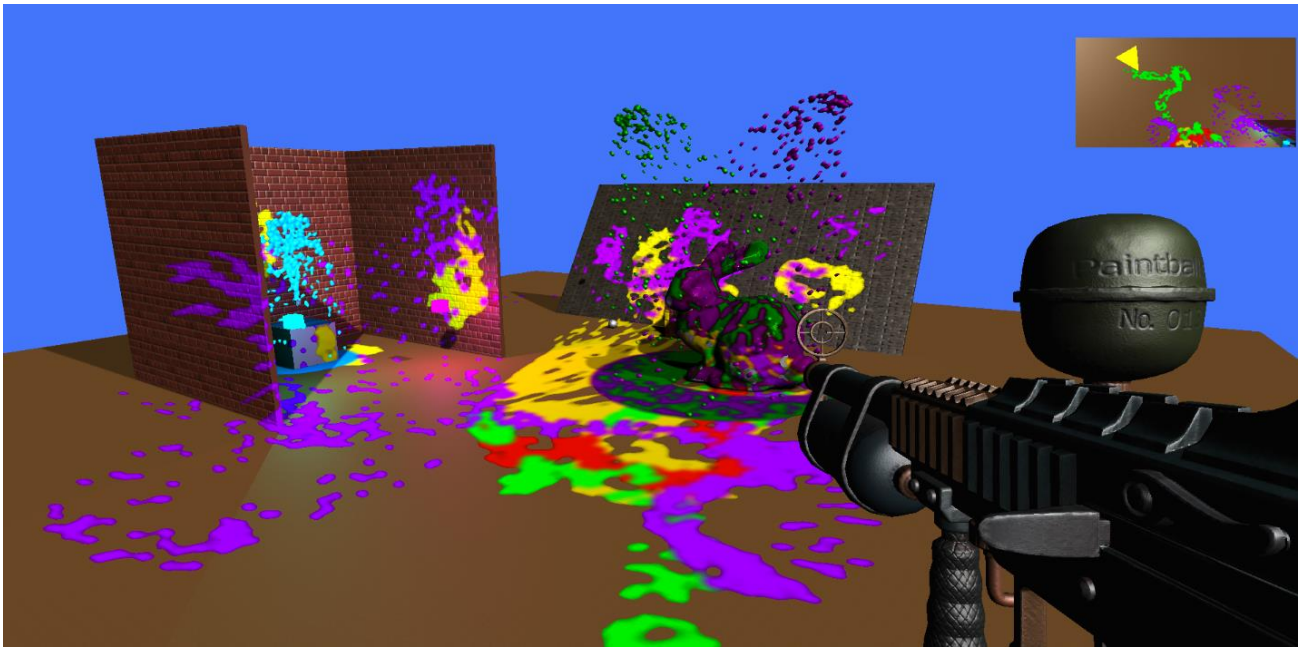
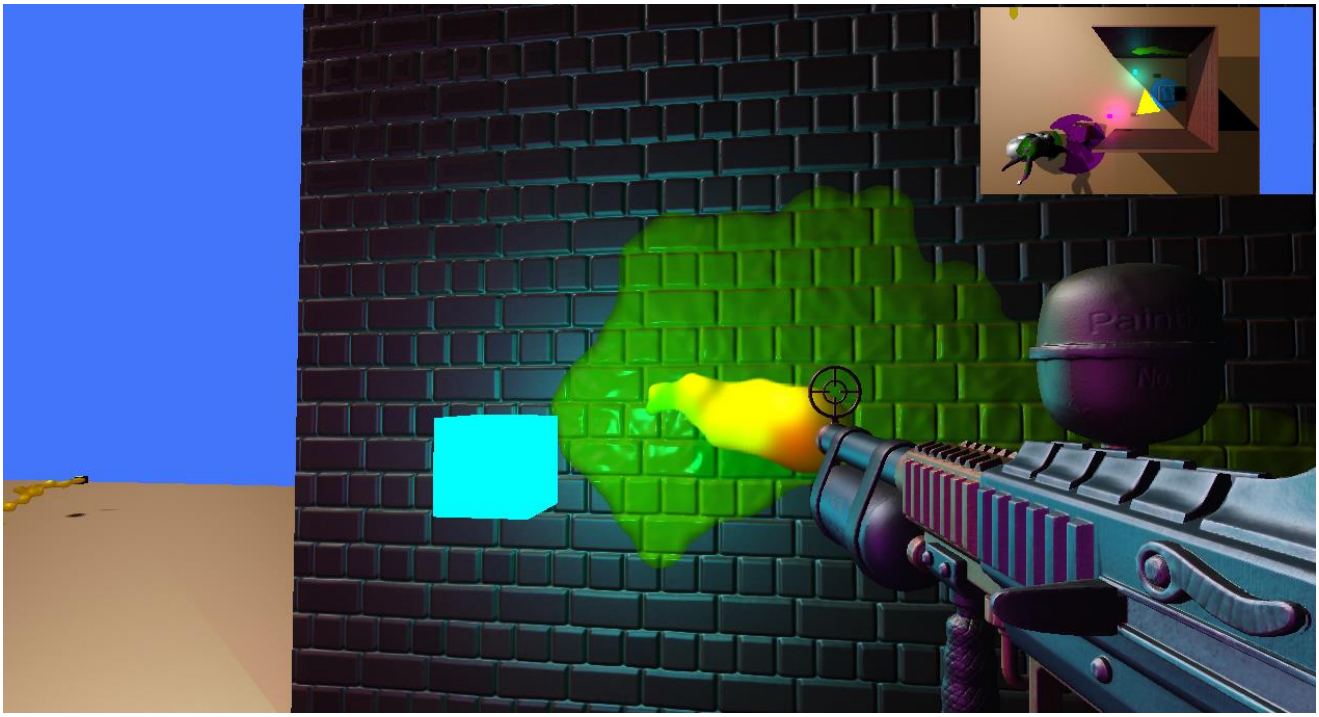
Chromancers

A simple paint-splatting 3D application





Luca Di Gennaro - 961208

Table of Contents

Introduction and goal

Painting in videogames is a very common practice, ranging from letting the player simply interact aesthetically with the environment, to becoming a downright game mechanic by itself.

An obvious example of the latter is the game Splatoon, a third-person shooter by Nintendo, where players' goal is to shoot colored ink to cover the most territory on the game level while confronting other players.

The goal of this project is to emulate the **ink shooting** mechanic and the consequential **painting effect** achieved on various surfaces in a 3D interactive application with OpenGL for rendering.

Features overview

Here, we will briefly introduce the two key features of the project: the ability to **shoot a "liquid-looking" stream of ink** and the ability to **paint on game entities** (such as textured models and meshes).

Shooting paintballs

The player has the capability to shoot projectiles to color the environment around them: each projectile is a physical object and can collide and interact with other entities in a dynamic world managed by a physics engine.

We want to make these projectiles look like a stream of ink droplets and the most straightforward way to do it is by using several spheres or similarly shaped round objects: we are going to call these **paintballs**.

This abstraction offers a few benefits as they:

- Resemble blobs of liquid quite well
- Are easy to simulate physically (including collision checking)
- Are easy enough to draw (especially when using low poly models and appropriate shading to hide rough edges)

However, it is easy to see that, in practice, they don't look like a cohesive stream of particles at all but single independent beads.

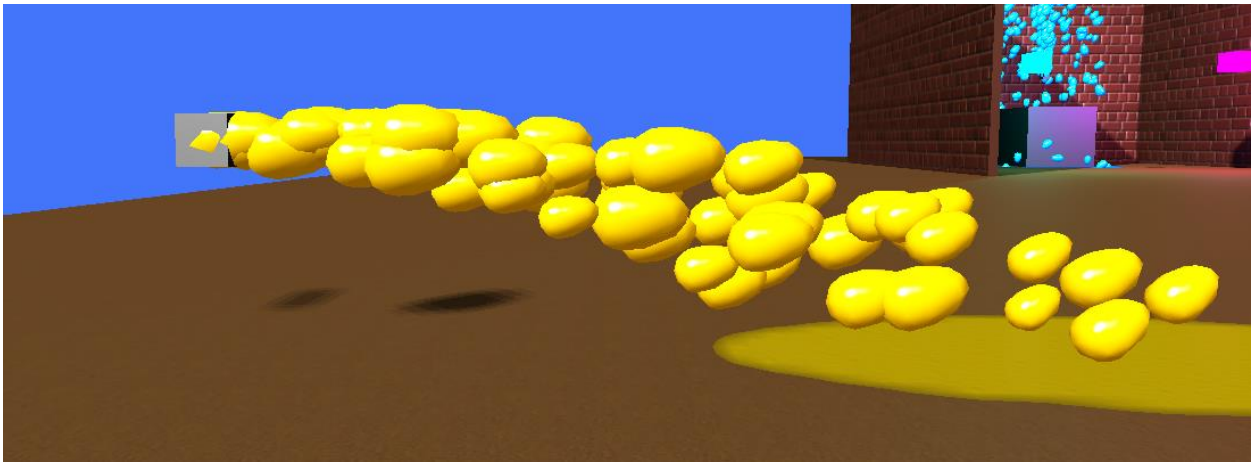


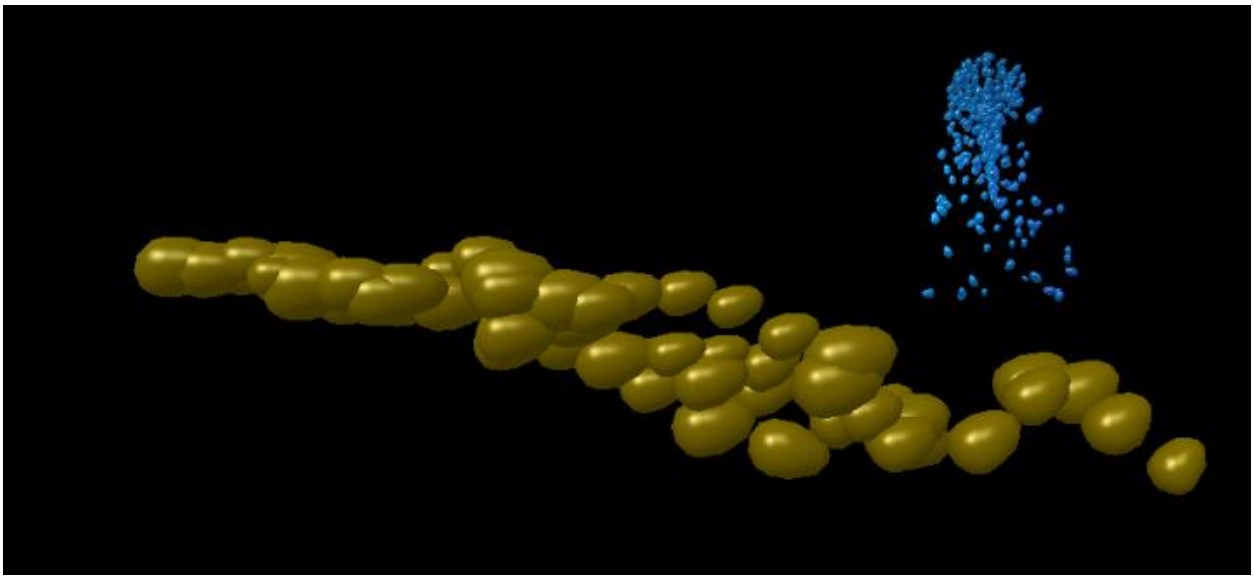
Figure 1 : First iteration of paintball generation

While researching how to achieve a better visual effect, we stumbled upon **meta-balls**: blobby objects with the ability to meld together when in close proximity.

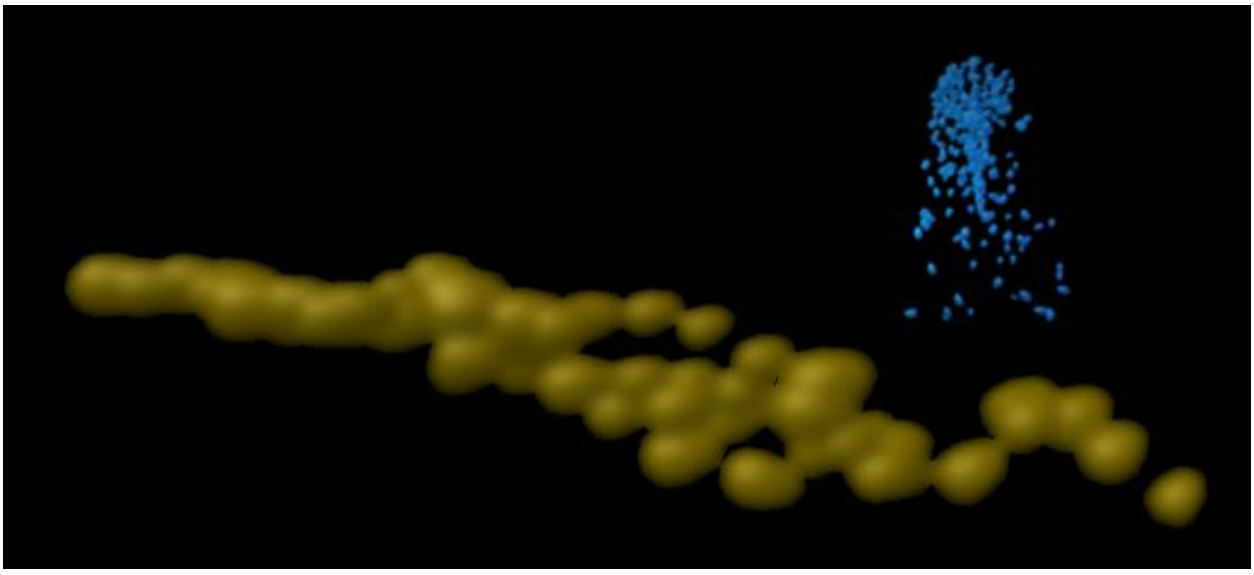
This looked promising on paper, since it approximates quite well the **"surface tension"** effect that liquids have; however, a quick investigation revealed that it would be a **computationally expensive** solution for our case, since we would have hundreds of paintballs to simulate at once in our scene.

Fortunately, we found another interesting idea: by using **screen-space effects** it is possible to achieve a similar result in a relatively **inexpensive** way.

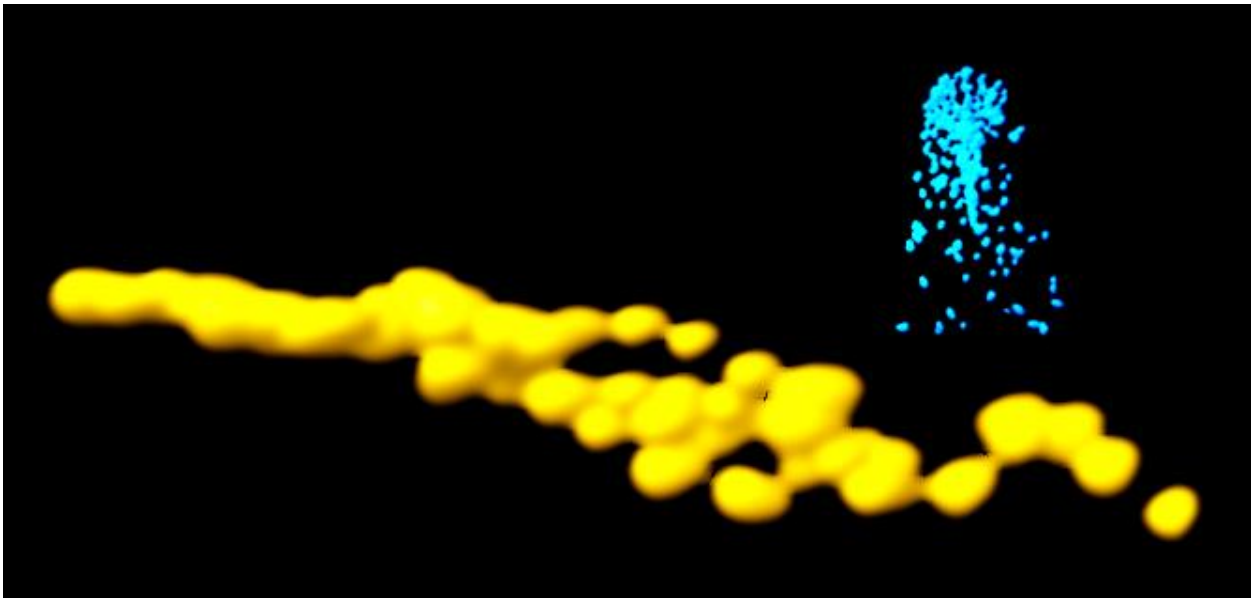
First, we want to draw all paintballs into a separate framebuffer so that we can operate freely on them.



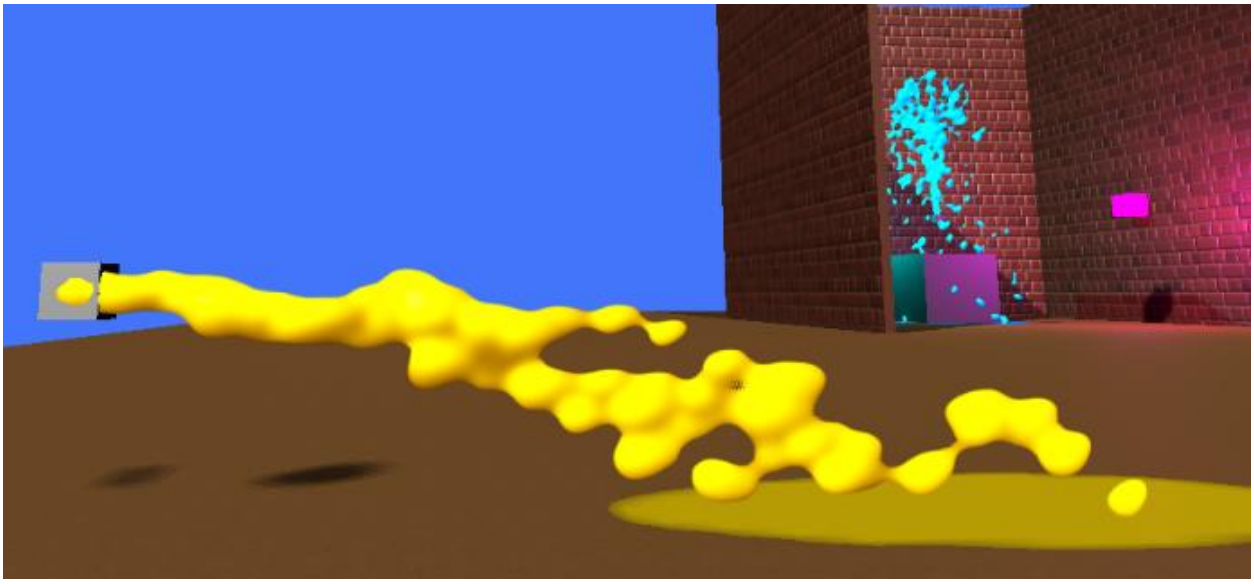
We want to apply some **blur** to the resulting image so that we can soften some edges here and there, amalgamating the single blobs together.



This already looks more organic since the blobs' borders expanded and merged with other neighbors: however, we lost some color detail, and the image appears blurry and less clear.



By **adjusting the color levels**, we can uniform some highlight and midtone hues so that they blend better together, as well as smoothing out some borders.



After merging this framebuffer back with the rest of the scene, this is the final result, which looks pretty close to what we wanted to achieve with meta-balls.

Painting on objects

Painting on an object can be done in several ways: some consider **painting the vertices** by altering their “color” attribute and interpolating it later in the fragment shader, others consider **painting in the UV space** of a texture linked to the object in question.

We decided to follow the latter approach because it is more **straightforward and modular**: modifying a mesh directly can be a hindrance, especially if it is shared between several game entities.

Furthermore, with the UV approach, we can **stack and blend** different texture layers together, achieving a richer visual effect, or having the possibility of painting in a non-destructive way on top of another texture.

In this project, each paintable object will have its own **paintmap**, which is a texture in which we bake all the color splats generated by impacts throughout the application runtime; when a paintball hits a paintable object, the latter will begin a “self-painting” procedure.

For this, we will use a technique akin to the one used for shadow maps: there, we would usually observe the scene from the light’s point of view and create a visibility map in light space by using the depth values.

In our case instead, we want to render the hit object from the impact point of view (in “**paint space**”), creating a projection volume to determine which fragments are going to be affected by the paint.

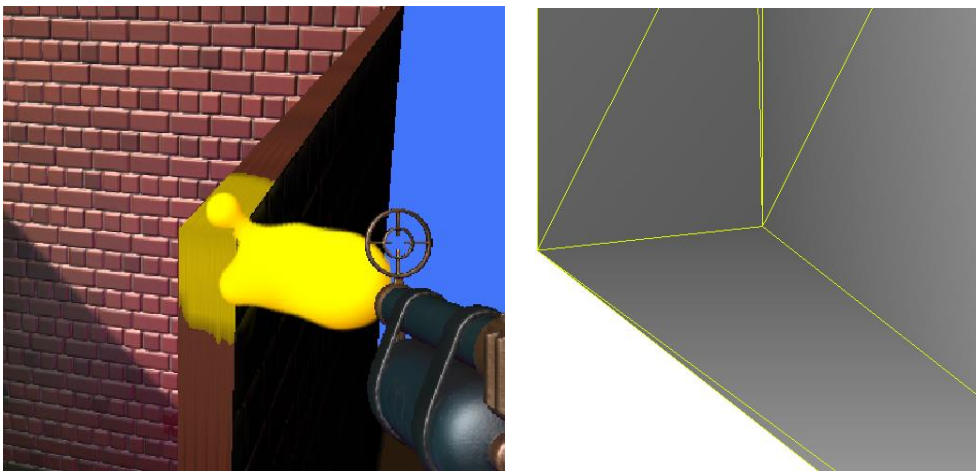


Figure 2 : Painted object and a depth map representation of the “paint space” projection, from the paintball’s point of view.

For each fragment in that projection, we are going to use their UV coordinates to store the new paint color value into the paint map, filtering it through a paint mask and blending it linearly with the previously baked color.

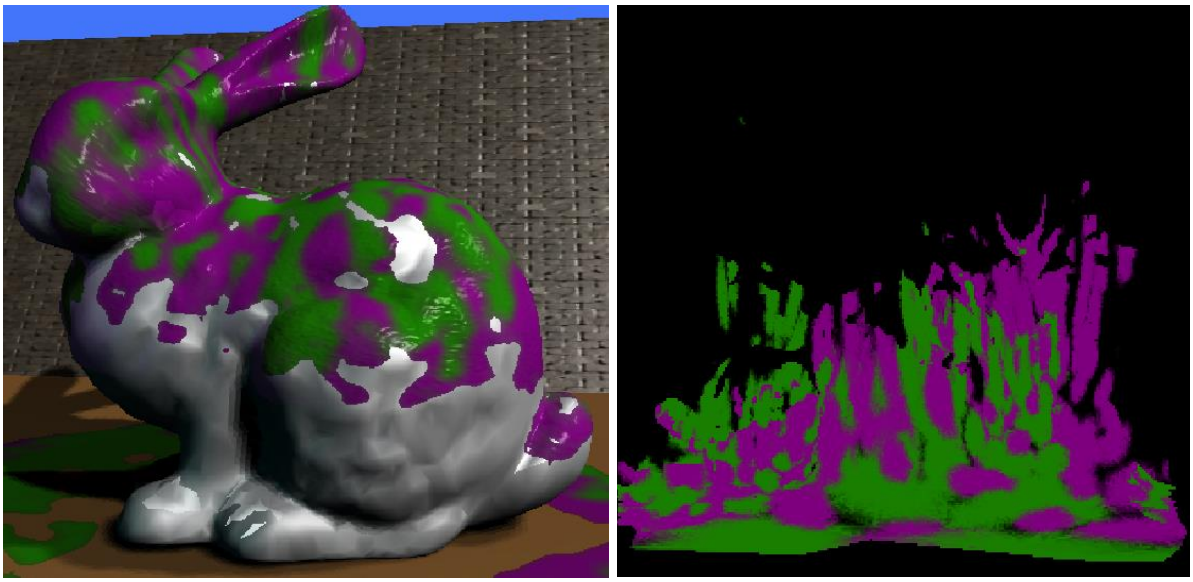


Figure 3 : Painted object and its relative baked and unwrapped paintmap.

We will see these steps in further detail in the implementation section, as well as how we can easily blend this paintmap on top of an already existing diffuse map.

Environment overview

The simulation depends on the following external libraries:

- OpenGL + GLFW, for visualization and user input purposes
- GLM, for vectorial computation
- Imgui + Implot, for the user interface
- Assimp + stb_image, for the assets loading
- Bullet, for the physics engine

The project is based on a basic self-made game engine, which elements are organized in the following sections:

- **Resources**, classes with the purpose of ***handling assets***
  - e.g. meshes, textures, shaders…
- **Scene**, classes which represent objects part of the ***game scene***
  - e.g. the player, entities, cameras, lights…
- **Components**, classes which modularly ***add behaviours*** to game entities
  - e.g. adding a rigid body, making an object paintable…
- **Utilities**, classes which interface with external libraries or are general purpose
  - e.g. physics manager, window wrapper, input manager, random generator…

We also have a collection of several shader files, some for lighting and texturing scene entities and others for shadow map computation or screen-space effects.

Implementation details

Since the codebase is very large, here we'll focus only on the **most relevant features** of the project; in any case, the code is well documented and with no shortage of comments to explain other details.

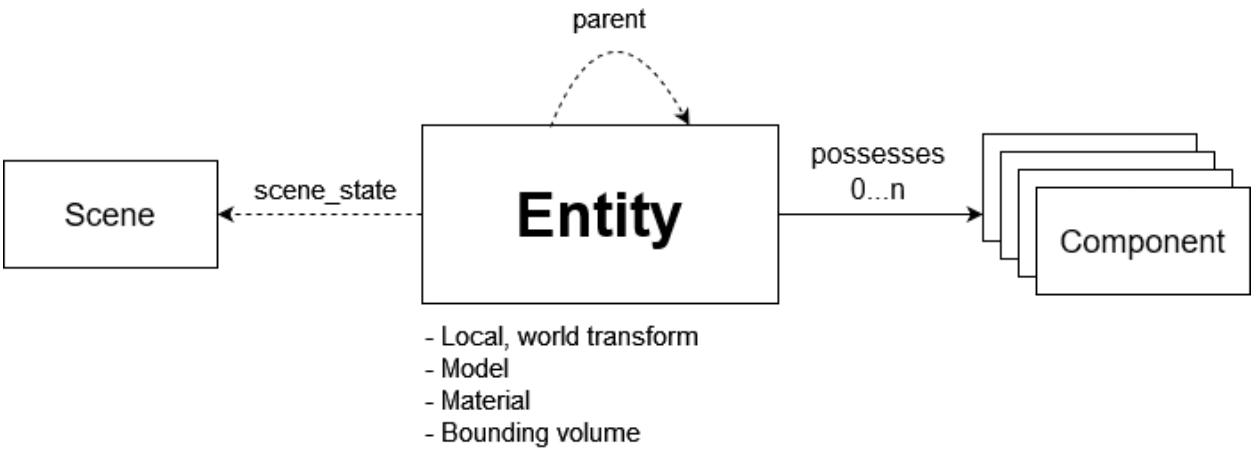
Engine elements

The main foundations of this application are the `Entity`, the `Scene` and the `Component` class.

Entity

chromancers/utils/scene/entity.h, entity.cpp

Our simulation world is primarily made up of entities, game objects which can be visualized and interacted with by the player.



Each entity can have a **drawable object** (a mesh/model) and a material that makes it possible to represent them visually in our scene. Additionally, a simpler bounding volume can be defined to help with culling operations.

An `Entity` can be brought into the world space by computing a **world transform** starting from the data contained in its local transform. Since each entity can have a parent, its world transform may depend on the parent's one, creating an transform inheritance relationship (useful for compound entities).

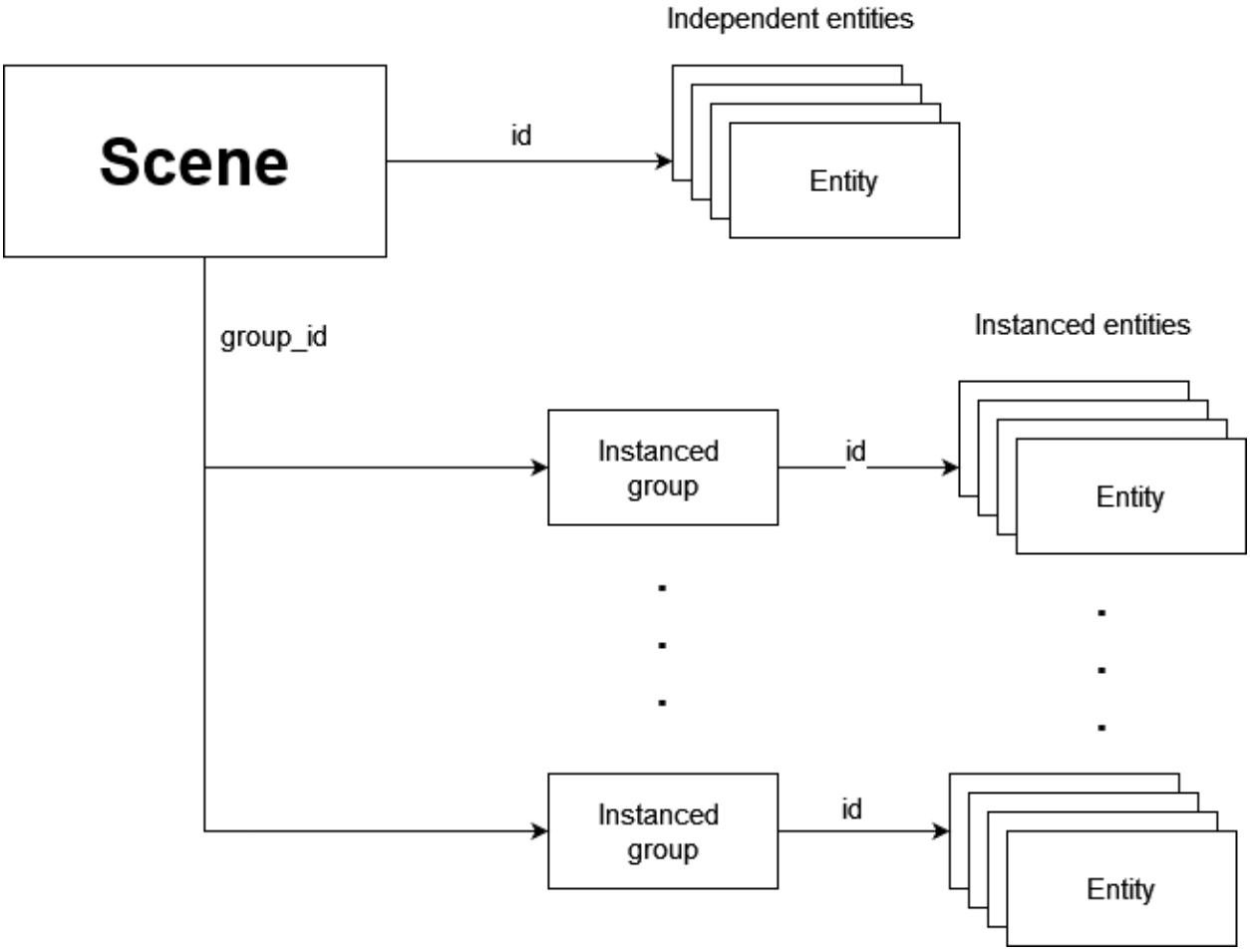
Entities can possess a **collection of components** which, as we will see later, represent a flexible way to add behaviour and functionality: the entity will need to initialize and keep it updated throughout the application lifetime so that they can function.

Each entity keeps track of the scene it is part of and by which id it is known as: this is useful to access common resources shared by the scene as well as communicating with it.

Scene

chromancers/utils/scene/scene.h, scene.cpp

Our entities will be contained and managed by the `Scene` class, which will make sure to initialize, update and draw all of them during the game loop.



The scene will distinguish between **independent** entities and **instanced** entities: since we will have to deal with hundreds of bullets, instanced drawing is necessary if we want to keep a serviceable frame per second amount.

Independent entities will perform a normal draw call, each using their own material and their own transform.

Instanced entities will be drawn together in an instanced draw call; since they will all share the same model and material (thus the same shader), it is possible to create different groups of instanced entities, each with its own configuration.

Each instanced group will load an array of **world transforms** to transform every instance into the shader: for this purpose, we used a Storage Shader Buffer Object (**SSBO**) to ensure a large enough memory block for these transforms.

```
// […] Material was bound for this group
for (auto& [id, instanced_entity] : instanced_group)
{
// Fill data about instanced group transforms
instance_group_transforms.push_back(instanced_entity->
   world_transform().matrix());
}

// Fill the shader's ubo/ssbo with the gathered transform data
utils::graphics::opengl::setup_buffer_object(instanced_ssbo,
GL_SHADER_STORAGE_BUFFER, 0, sizeof(glm::mat4),
instance_group_transforms.size(), GL_DYNAMIC_DRAW,
glm::value_ptr(instance_group_transforms[0]));

// Perform the instanced draw on the common model of the group
instanced_group.begin()->second->model->
draw_instanced(instance_group_transforms.size());
```

Figure 4 : Excerpt of the instanced group drawing method

By keeping track of the current camera in usage, the scene can also perform a basic frustum culling check to avoid drawing entities that are not into the camera's frustum.

Components

chromancers/utils/component.h

The `Component` class is the basic building block for adding functionality to an `Entity`, which follows the composition over inheritance paradigm.

With components we can add **additional behaviors** to our entities, modularly and independently.

Each component will have a reference to its **parent entity** so that it is able to influence it according to its purpose: for example, a `RigidBodyComponent` would keep the parent entity's transform in sync with its counterpart in the physical engine.

Here is a summary of all component types in our project:

- `RigidBodyComponent`, keeps track of a `RigidBody` object handled by the Bullet Physics engine and syncs the parent entity transform to it
- `PaintballComponent`, simulates a projectile, triggering a painting event on the affected surface and self-destructing on collision
- `PaintableComponent`, adds a paintmap to an entity that will be altered by painting events
- `PaintballSpawnerComponent`, gives an entity the capability of spawning paintballs by creating and updating a `PaintballSpawner` object

Physics and Collision handling

The scene simulation runs along with a **physical simulation** managed by an implementation of a `btDiscreteDynamicsWorld` (from the Bullet library).

We can create rigid bodies with different kinds of **colliders**, such as a sphere, a collider or even a convex hull. During their creation, we can also define a **pointer to user data**: we will use this to link the physical body to an Entity object.

Each rigid body can define a **collision filter** to avoid collisions with certain groups of bodies, according to the defined bit mask: for example, this feature is used by paintballs to not collide with other paintballs.

Collision detection is performed by iterating through all contact manifolds provided by the dynamics world: these manifolds are structures containing information about all contact points between pairs of bodies that collided during the physics update step.

We want to retrieve all kinds of information about the contact points so that we can send them to the involved entities for processing: every entity has a `on_collision()` function **callback** that will use that data to resolve the collision or react to it.

```
// Iterate through all manifolds in the dispatcher
for (int i = 0; i < numManifolds; i++)
{
   // A contact manifold is a cache that contains all contact points between
pairs of collision objects.
   btPersistentManifold* contactManifold = dynamicsWorld->getDispatcher()->
getManifoldByIndexInternal(i);
   const btCollisionObject* obA = contactManifold->getBody0();
   const btCollisionObject* obB = contactManifold->getBody1();

   // Retrieve the user's objects (entities) that have collided
   UserObject* ent_a = static_cast<UserObject*>(obA->getUserPointer());
   UserObject* ent_b = static_cast<UserObject*>(obB->getUserPointer());

   for (int j = 0; j < numContacts; j++)
   {
      // […] Collect info about collision

      // Forward collision info to entities via callback
      ent_a->on_collision(*ent_b, glm_ptA, glm_norm, glm_impls);
      ent_b->on_collision(*ent_a, glm_ptB, glm_norm, glm_impls);

   }

}
```

Figure 5 : Excerpt of the detect_collision() function in the physics engine

Components can also have an `on_collision()` callback propagated to them by the parent entity so that they too can react accordingly. As we will see, this is how paintballs make paintable objects aware of collisions.

Paintball implementation

A paintball is an instanced `Entity` that possesses a `RigidBodyComponent` and a `PaintballComponent`.

A paintball usually has a **limited lifetime**: it is either destroyed by the scene after an impact or after a set amount of time (to avoid paintballs drifting infinitely into space and taking up computational resources).

Collisions

On impact, the `on_collision()` callback is triggered by the `PaintballComponent`: if a paintable entity (an Entity with `PaintableComponent`) was struck, then a pair of view-projection transforms are computed from the paintball's point of view.

```
// Create paintspace viewprojection
glm::mat4 paintProjection = glm::ortho(-frustum_size, frustum_size, -
frustum_size, frustum_size, paint_near_plane, paint_far_plane);

glm::mat4 paintView = glm::lookAt(paintball_position - paintball_direction *
distance_bias, paintball_position + paintball_direction, paintball_up);

// Make the paintable entity aware of the paintball collision and let it update
its paintmap
other_paintable->update_paintmap(paintProjection * paintView,
paintball_direction, paint_color);
```

Figure 6 : Excerpt of the on_collision() callback in PaintballComponent

This pair will be sent to the paintable object so that it can use it to compute the projection volume in **paintspace** and update the paintmap accordingly.

Generation

Paintballs are easily generated in numbers by paintball spawners (powered by the `PaintballSpawner` class).

```
Model*    paintball_model    {nullptr}; // Model
Material  paintball_material; // Material

glm::vec4 paint_color{ 1.f, 0.85f, 0.f, 1.f }; // Paint color

float paintball_weight { 1.0f }; // Paintball rigidbody mass
float paintball_size   { 0.1f }; // Paintball size
float size_variation_min_multiplier { 1.f }; // Min value of size multiplier
float size_variation_max_multiplier { 1.f }; // Max value of size multiplier

unsigned int rounds_per_second { 100  }; // N. of paintballs spawned per second
float        shooting_speed    { 20.f }; // Force applied to spawned paintball
float        shooting_spread   { 1.5f }; // Deviation from the aimed direction
```

Figure 7 : The various parameters used to customize the paintball spawn system

Given all the necessary generation parameters (such as rate of fire, paintball visuals, physical properties…) we can generate a **steady stream of paintballs**.

During generation, paintballs can have slightly **random variations** of size, speed and spread: this is done to make the stream less uniform thus more believable.

Each paintball spawner will share the **same material**: this means that we can use the instanced group feature of the `Scene` class to draw together paintballs of the same spawner with the same properties (e.g. color).

An `Entity` can encapsulate the paintball spawning behavior by attaching a `PaintballSpawnerComponent` to it: the entity's transform position and forward direction will be automatically used as spawning parameters. This was useful to set up several fountains of ink in the scene as well as generating paintballs while in motion (e.g. the player's gun).

Shading effect

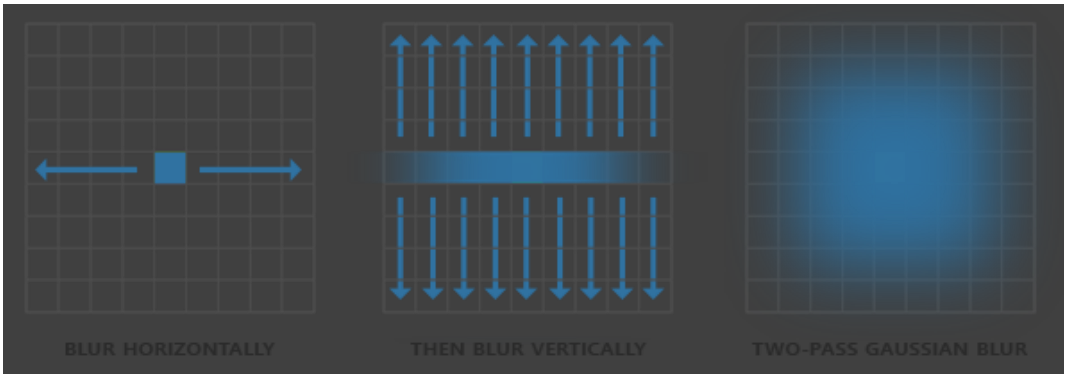
When paintballs are drawn, we do not draw them in the default framebuffer but in a **specific framebuffer** for only paintballs (`paintballs_framebuffer`). The same is true for the rest of the scene (`world_framebuffer`).

The color attachment format of the `paintballs_framebuffer` is RGBA so that we can also work with transparency and is cleared each frame with (0,0,0,0). The relative depth attachment will be useful later when we need to perform depth testing for paintballs.

The following shader effects will all operate on dedicated postfx framebuffers.

Blur

We start with the blurring effect (*paintblur.frag*) on the rendered paintballs image; we use a **two-pass gaussian blur** technique, since a gaussian filter is separable linearly and lets us reduce the number of computations needed to compute the blur.



```
vec4 result = texture(image, TexCoords) * weights[0]; // current fragment's
contribution

if(horizontal)
{
   for(int i = 1; i < 5; ++i)
   {
result += texture(image, TexCoords + vec2(tex_offset.x * i, 0.0)) * weights[i];
result += texture(image, TexCoords - vec2(tex_offset.x * i, 0.0)) * weights[i];
   }
}
else
{
   for(int i = 1; i < 5; ++i)
   {
result += texture(image, TexCoords + vec2(0.0, tex_offset.y * i)) * weights[i];
result += texture(image, TexCoords - vec2(0.0, tex_offset.y * i)) * weights[i];
   }
}
```

Figure 8 : The two-pass blur implementation

However, we noticed that paintballs farther from the camera were being blurred so much that they almost disappeared at long distances; at the same time, paintballs closer to camera were not blurred enough.

For this reason, we used the depth attachment from the `paintballs_framebuffer` to estimate the distance of each paintball from the camera: the depth image is not linear, so we need to linearize it first.

```
// linearize depth value to use depth as a way to tell distance
float linear_depth = LinearizeDepth(texture(depth_image, TexCoords).r) +
depth_offset;

// sample depthmap, the deeper the fragment, the less it is blurred
float depthbased_blur_strength = blur_strength / linear_depth;

// gets size of single texel
vec2 tex_offset = depthbased_blur_strength / textureSize(image, 0);
```

Figure 9 : Depth-based blur strength implementation

Once computed, we can use this distance to influence the texel offset used during the blurring process: the bigger the offset, the blurrier the fragment. This fixed the said problem very well.

In the end, we repeat these passes (horizontal + vertical) for a number of times specified by the user, using “ping-pong” framebuffers to store the partial results.

Smoothstep

After the blur is completed, we want to make the image crisper, with sharper outer edges and softer, uniformed inner colors.

To do so we sample the final blurred image and interpolate the color and alpha values by using the smoothstep function, regulating the image **color levels** and adjusting the hues' highlights and midtones.

```
vec3 s_color = sampled.rgb;
float alpha = sampled.a;

s_color = smoothstep(t0_color, t1_color, s_color);
alpha = smoothstep(t0_alpha, t1_alpha, alpha);

FragColor = vec4(s_color, alpha);
```

Figure 10 : Smoothstep interpolation

The user can control this process by altering the threshold values of the function, so that this effect can be adjusted as needed (e.g. adapt to the scene lighting).

Merge

Finally, we need to merge the post-processed paintballs with the rest of the scene: to do so, we use once again the depth attachment from the `paintballs_framebuffer` and test it against the `world_framebuffer`.

```
// Blend the colors by using the alpha of the least deep sample
if (depth0 <= depth1)
{
   // color0 has precedence since it has smaller depth
   final_color = color0 * color0.a + color1 * (1 - color0.a);
}
else
{
   // color1 has precedence since it has smaller depth
   final_color = color1 * color1.a + color0 * (1 - color1.a);
}
```

Figure 11 : Merging framebuffers based on depth

Depending on the sampled depth, we choose which color goes on top of the other. After that, we consider their alpha and interpolate linearly between the top fragment and the bottom fragment.

After this process, we managed to merge the paintballs and world color attachments into one image that can be blitted into the default framebuffer and shown to the screen.

Paintables implementation

A paintable object is an Entity that possesses a `RigidBodyComponent` and a `PaintableComponent`.

The `PaintableComponent` makes an object paintable by owning a **paintmap**, a texture that persistently holds information about paint impacts happened throughout the entity's lifetime.

The paintmap is generated as a blank texture (RGBA 0,0,0,0) and is populated via the `update_paintmap()` function: this callback is invoked by a paintball when it strikes a paintable object.

We retrieve information about the computed paintspace matrices, the impact direction and the paint color, passing them to a shader (*texpainter.vert, .frag*) which will:

- Transform the vertices to paint space
- Load the previous color of the paintmap at the fragment's UV coordinates
- Interpolate the old, sampled color with the new paint color, if the fragment passes a splat mask test and an incidence test
- Save the updated color at the same UVs in the paintmap

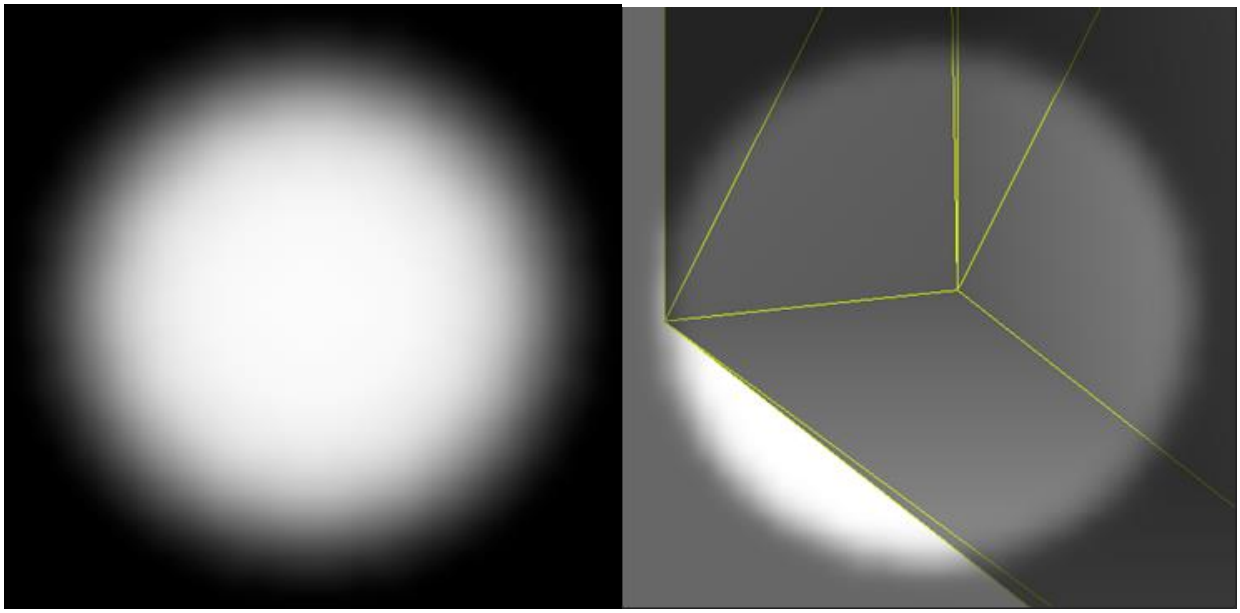


Figure 12 : Usage of the splat mask, which filters which fragments will be affected by paint in the paintspace projection

Now that we know how to keep the paintmap updated, we need to understand how to apply it to objects.

The `Material` class holds several uniforms and textures that may describe an object: initially we just supported a diffuse and a normal map, but this was not enough if we wanted to add the paintmap layer on top of a textured object.

The solution was to add a detail map pair:

- The **detail diffuse map** would contain the paintmap and would be layered on top of the already existing diffuse map
- The **detail normal map** would contain a normal map just for the paint, mixed with the object's normal map
    - This is done to achieve a different surface lighting appearance for painted areas of an entity, making it look "wet"

```
if(sample_detail_diffuse_map == 1 && sample_detail_normal_map == 1)
{
   if(detail_diffuse_color.a > detail_alpha_threshold)
   {
      float ft = detail_diffuse_color.a;

      // Update normal considering the detail normal map
      N += calculateNormal(detail_normal_map, sample_detail_normal_map,
fs_in.twNormal, finalTexCoords) * detail_normal_bias * ft;

      // Update surface color considering the detail diffuse map
      surface_color = mix(surface_color, detail_diffuse_color *
detail_diffuse_bias, ft);

      // Update fragment's shininess where detail map (paint) is visible
      float detail_shininess = 512.f;
       shininess_factor = mix(shininess_factor, detail_shininess, ft);
   }
}
```

Figure 13 : Shader excerpt (from default_lit.frag) on how the detail maps influence the final color and lighting calculation

We updated the material specification in the principal lit shader (*default_lit.frag*) so that it can support the paintmap visualization and layering.

Additional features

Other than the explained main features, we wanted to study the implementation of some other visual effects.

Parallax mapping

We wanted to give more depth to bumpy surfaces like brick walls, making it look like the paint could seep through their nooks and crannies: to do so, we looked into parallax mapping.

By giving each object a **displacement map**, we can easily make it look like a fragment is deeper or closer to the camera by altering the texture coordinates based on the view direction.

The relative code is in the `CheapParallaxMapping()` function, in the *default_lit.frag* shader.

Point light shadowing

We wanted to investigate how hard it would be to achieve shadowing from point lights and we observed how problematic it is.

For starters, a single light space projection is not enough since a point light is omnidirectional and thus casts shadows all around it.

Since a projection is a rectangle, we could create a projection for each perpendicular direction of the light: forward, backwards, upwards, downwards, left and right. We essentially need to create a rectangular polyhedron of projections or, simply, a cube.

So, instead of a simple texture for a shadow map, we need a **cube map** and OpenGL lets us do that via the `GL_TEXTURE_CUBE_MAP` definition.

This means that, for each point light, we should redraw the scene six times, once for each face of the cube. This is what makes point light shadows very costly in terms of performance.

An alternative to that is to use a geometry shader with the final goal of rendering all the faces in a single graphics pass. Starting from the vertices of the scene, the geometry shader's job is to emit new vertices to replicate them along the six faces of the cube dynamically. For each face of the cube, we will have a different matrix to transform the vertices into light space.

With this data, the shadow map will be computed by measuring the distance of each fragment from the light source.

The relative code is contained in the `PointLight` class (*chromancers\utils\scene\Light.h*) and its shader implementation is in the *shadow_cube* shaders.

Performance

Hardware

The simulation testbed is composed by a Nvidia GeForce RTX 3060 coupled with a Ryzen 5 3600 and 32GBs of RAM @ 3200 MHz.

Setup

We evaluated the simulation with four static spawners generating each a certain number of paintballs. The simulation is rendered at a resolution of 1280x720.

We have two point lights and two directional lights.

All entities in the scene are paintable (thus have a paintmap) except from

- A sphere acting as a control entity (still has a rigid body)
- The paintball spawner entities
- The player's gun
- The point light cubes

Analysis

Paintballs generated live until a collision happens or their lifetime expires, thus performance may depend on the paintball spawner position and configuration.

For this reason, we will consider the number of total paintballs in the scene as the effective metric.

Paintballs per second per spawner	Effective paintballs in scene	Avg. FPS
0	0	~800
100	~450	~600
500	~2700	~100
1000	~5000	~30

Instanced drawing was crucial for keeping a reasonable fps rate given the number of paintballs present in the scene.

Generally, to achieve an acceptable looking stream of ink, we need to generate an amount ~100 paintballs per second per player, which is a good compromise between quality and performance.

Conclusions

All things considered, even though there are still possible margins of improvement and optimization, we are satisfied with how good the ink simulation looks like for the performance price it brings, making these effects serviceable for a real-time application like a videogame.