Real Time Graphics Programming

Emmanuele Villa

2021/2022

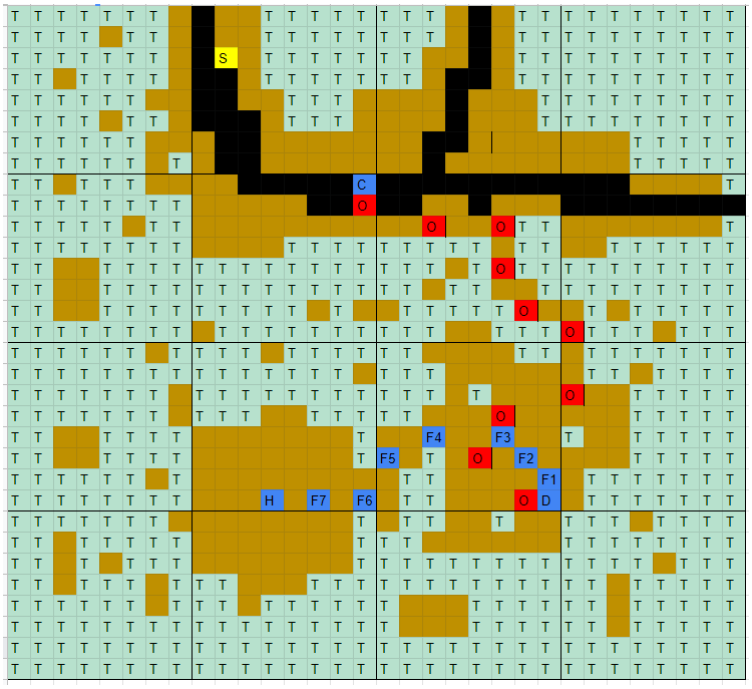
Contents

1 Introduction

This project aims to replicate some visual effects from the game "The Witcher 3", referred to "witcher senses" in the game.
In the application, one can activate the witcher senses by keeping the space bar pressed, and it will enhance particular objects or show a trail to be followed.
Since this is a third-person game in an out-world map, a collision system has also been implemented to avoid player interpolation with objects and to move the camera away if there's an object obscuring the player.
The algorithms pseudo-code presented in this document is a simplification of the true algorithm for the sake of reading and comprehension.

2 Creating the level map

The level map is semi-parameterizable, meaning that there's a csv file embedded in the code that describes the map, objects and trails. One can freely modify the map with some constraints like the map size and the maximum numbers of tree objects present.



During the application initialization, the map file is parsed and all the static content is generated before the first render pass, like:

- The model matrices of the static objects and their AABBs
- The points composing the trails (odors and footprints)
- The AABBs tree hierarchy used to speed up calculations

All these operations are executed in a loading render loop that just renders a black screen. Those operations are split in more frames to avoid blocking the frame rate during calculations, so that if a loading animation would be inserted it wouldn't stutter.
In the code, those steps are made explicit using an enumerator describing the current app state:

```
enum class AppStates {
    LoadingMap,
    LoadingAABBs,
    CreatingAABBsHierarchy,
    InterpolateOdorPath,
    CreatePaths,
    Loaded
    };
```

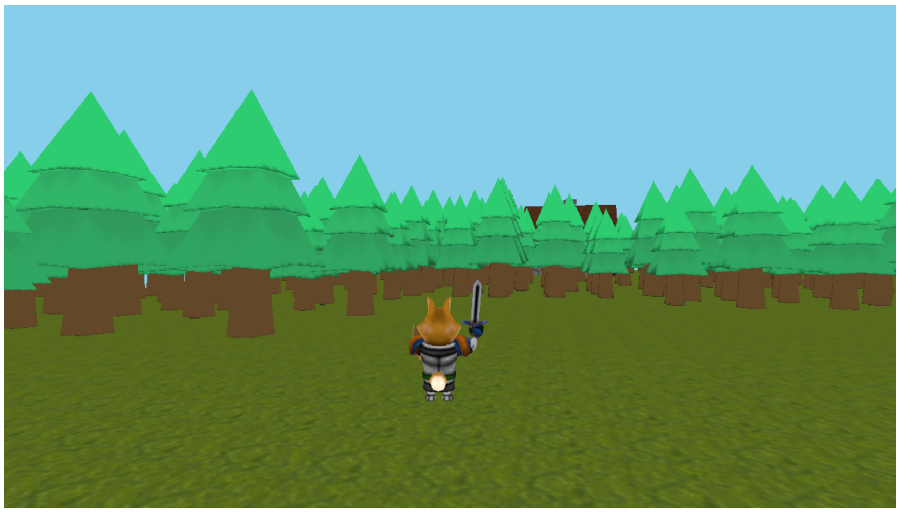
3 3d Model creation

Since the map has a lot of trees, it's not feasible and useless to draw them one by one by sending the triangle information to the GPU every time.
To draw the trees we'll use the method glDrawElementsInstanced(...) that lets us send the triangles only once to draw the same model multiple times.
Also, since the matrices are too many we can't send the information as we do with a "standard" uniform object, but we have to use a buffer that contains the data.
The algorithm used is the following:

- Load the tree model mesh
- Create a list of matrices with the transformations of each tree
- Pass the list using a Uniform Buffer Object
- Call glDrawElementsInstanced() with the number of objects
- Access the correct matrix in the vertex shader by using the gl_InstanceID index that comes with it

In this way we can achieve a great number of trees in the scene without occluding the bus:

```
subroutine(vertshader)
vec4 instanced() {
    return projectionMatrix
        * viewMatrix
        * modelMatrices[gl_InstanceID]
        * vec4(position, 1.0);
}
```



4 Player Movement

The player can only move along the X and Z axis, and rotate around the Y axis, meaning that its position can be described using these three values: posX, posZ and rotationY.
When the player presses W, A, S or D, the posX and posZ values are updated accordingly. The rotationY changes when the player moves the mouse while pressing the mouse right button.
The Y rotation is always possible, as the player collider is described by a squared AABB that doesn't change when the player rotates.
On the contrary, the movement along the X and Z axis can be blocked by obstacles like trees and so a collision must be checked to see if that movement is allowed.

4.1 Collision handling

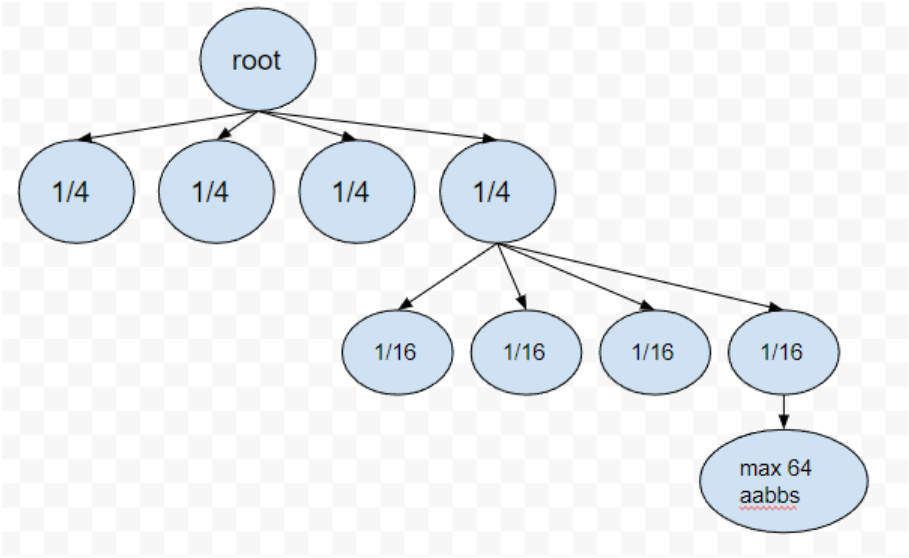
4.1.1 Spatial subdivision

During the application loading, all the objects in the scene are grouped in a tree of AABBs in the following way:

- As root of the tree, an AABB with the same size as the map is created
- Next, we split that AABB in 4 sub-AABBs of equal size and add them as its children
- The same operation is executed for those 4 AABBs, resulting in a tree with 1 AABB in the first level, 4 in the second level and 16 in the third
- For each object in the map, we add it as child of all the third-level AABB that collides with it, using the AABB vs AABB collision check

Note that:

- The map is a 32x32 square, so the third-level AABBs describes a portion of 8x8, meaning that they can have a maximum of 64 children
- In case an object collides with 2 AABBs (it could happen because trees are randomly displaced in their cell), it is added to both
- All objects collides with every object in the Y axis, so the collision check only happens on the X and Z axis



4.1.2 Collision detection

To check the collision, we use following algorithm that takes an AABB as input and check if it collides with itself or one of its children:

```
bool checkCollision(other) {
    if I don't collide with 'other' on X, return false
    if I don't collide with 'other' on Z, return false
    if I am a tree leaf, return true
    if I have no children, return false
    return true if any of my children.checkCollision(other) returns true
}
```

Checking the player collision now it's just a matter of calling:

```
aabbRoot.checkXZCollision(playerAABB)
```

4.1.3 Collision response

I could interrupt the movement as soon as the player collides, but this approach has a con: the player would completely stop. This doesn't feel natural, because it should happen only if the player is trying to walk into that object perpendicularly to one of its axis.
What should happen, and what happens in all games, is that the player must be blocked only in the direction that collides with the object, and not also in the other. In this way, when the player bumps into an object, it slides in the free direction instead of staying completely still.

To achieve this result, the following algorithm is implemented:

```
oldPosX = X position in the previous frame
oldPosZ = Z position in the previous frame
posX = X position in this frame
posZ = Z position in this frame

playerAABB = AABB(posX, posZ)

if playerAABB doesn't collides with objects, return
otherwise I check for collision when moving only on X and only on Z

collisionOnX = aabbRoot.checkCollision(AABB(posX, oldposZ))
collisionOnZ = aabbRoot.checkCollision(AABB(oldPosX, posZ))

if collisionOnX {
    posX = oldPosX
}

if collisionOnZ {
    posZ = oldPosZ
}
```

During my tests, this algorithm took an average of 50 microseconds when the player doesn't collide with anything, and 150 when it collides, which makes sense since we check the hierarchy only one time in the first case and three times in the second one.
The basic implementation of checking each AABBs took around 200 microsecond for the first case and 600 for the second, resulting in a speed up of 4 times with only three levels in the tree. Other improvement could be achieved by splitting the map in another 4x hierarchy level.

5 Camera

The camera follows the player at a fixed Y height, and so the camera can be described with just one additional value: the distance from the player.
The X and Z position of the camera is then calculated like this:

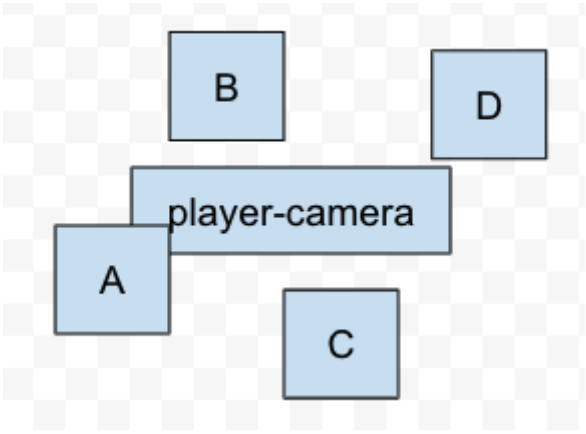
```
rotationY = the player rotation
distance = distance from the player to the camera
posX = the X position of the player
posZ = the Z position of the player
GLfloat distX = -sin(glm::radians(rotationY)) * distance;
GLfloat distZ = cos(glm::radians(rotationY)) * distance;
GLfloat cameraX = posX + distX;
GLfloat cameraZ = posZ - distZ;
return glm::vec2(cameraX, cameraZ);
```

5.1 Collision checking

This formula calculates the position vector of the camera, but as for the player, we have to check if the camera collides with something or -more generally- if there's any object between the camera and the player.

5.1.1 Broad-phase collision detection

At first, a broad-phase check is executed to discard the objects that aren't for sure in the player-camera line. To do so, we create an AABB using the player and camera coordinates and check for collision against the same AABB hierarchy used for the player.
In the following picture, only the object A could collide with the line and all the other objects are discarded in this step because if they don't collide with the AABB they can't collide with its diagonal:



5.1.2 Second broad-phase collision detection

Before checking the collision with the line, there is another step that speeds up: checking if the camera point collides with the remaining AABBs. Since the camera starts in free space and simply moves, in the majority of the cases it's the camera that will directly collide with an object; the only exception would be an object that enters in the line between the camera and the player, but due to the nature of the map and the camera system this is way less likely.
So, in this step we simply check:

```
bool fasterCheck =
    MinX <= camera.x
    && MaxX >= camera.x
    && MinZ <= camera.y
    && MaxZ >= camera.y;
```

If one of the AABBs collides in this way, I can say that there's an object without needing to advance with the calculation. Otherwise I still pass all the AABBs to the next step

5.1.3 Mid-phase collision detection

In this last step I need to do the real AABB-Segment check to see if any point in the camera-player segment collides with one of the AABB.
To do this I:

- Calculate the equation of the line passing through the camera and the player in the y = mx + c form
- Substitute the minX, maxX, minY and maxY AABB values in the equation to find the intersection points
- If one of this points is inside the AABB, then it collides with the line

5.1.4 Collision response

When we find an object occluding the camera, we have to apply a solution: the general solution in games is to move the camera closer to the player until it's occluded no more.
We don't want to show any frame where the camera is blocked, so we must move the camera closer and closer before rendering the frame until it collides no more: since it doesn't collide with the player and the player doesn't collide with anything, we can be sure that at some point we will reach that condition.
At the contrary, since we don't need to abruptly move the camera back when there is free space, we can avoid to waste calculation time and just animate the camera going farther.
To achieve this, we decide a delta value and do the following:

```
vec2 currentCamera = buildCameraPos(distance)
bool currentCollides = checkCollision(currentCamera)

if current doesn't collide and distance is less than max distance {
    vec2 fartherCamera = buildCameraPos(distance + delta)
    bool fartherCollides = checkCollision(fartherCamera)
    if farther doesn't collide {
        distance += delta
        currentCamera = buildCameraPos(distance)
```

```
    }
} else if current collides {
    do {
        distance -= delta
        nearCamera = buildCameraPos(distance)
    } until nearCamera doesn't collide
    currentCamera = nearCamera
}
```

6 Witcher sense

6.1 Generic effects

While the witcher sense are active, three effects are always present: the camera approaches the player, the scene becomes more grey, especially at the sides, and there's a pincushion distorsion.

6.1.1 Camera distance

Moving the camera closer to the player is just a matter of decreasing the maxCameraDistance and adding a check before applying the previously described camera algorithm:

```
if(cameraDistance > maxCameraDistance) {
    cameraDistance = maxCameraDistance;
}
```

6.1.2 Grey scene

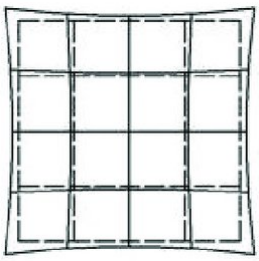
This effects darkens the whole scene and additionally darkens the right and left side, and it's applied in the fragment shader:

- Add a vec3(value) to all the fragments color
- Take the parable x^2+x, with y=0 in x=-1 and x=0
- Raise it by 0.1, to have y=0 in the points x $\sim$ -0.9 and x $\sim$ -0.1 instead
- Shift the uv.x value by -1, to have it in the range (-1,0) instead of (0,1)
- Calculate the y parable value using the shifted uv.x
- Clamp the resulting y in the range (0,1)
- Add another vec3(y) to the fragment color

You can see the result in the next section, as this is done along with the pincushion distorsion.

6.1.3 Pincushion distorsion

To apply the pincushion distorsion to the whole scene, we need an additional render pass.
In the first pass, the scene is rendered to a frame buffer with an attached texture. In the second pass, the texture is applied to a plane in front of a camera and the fragment shader applies the distorsion.



The code of the pincushion distorsion, alongside with the image darkening, is the following:

```
subroutine(fragshader)
vec4 pincushion() {
    vec2 repeatedUV = mod(interp_UV * repeat, 1.0f);
    float newX = interp_UV.x - 1.0f;
    float y = pow(newX, 2) + newX + 0.1;
    y = clamp(y, 0.0f, 1.0f);
    vec2 uv = interp_UV.xy - vec2(0.5, 0.5);
        float uva = atan(uv.x, uv.y);
    float uvd = sqrt(dot(uv, uv));
    uvd = uvd * (1.0 + distorsion * uvd * uvd);
    vec3 col = vec3(texture(tex, vec2(0.5) + vec2(sin(uva), cos(uva)) * uvd).xyz);
    return vec4(col + vec3(distorsion/20.0f) + vec3(distorsion * y), 1.0f);
}
```

And here is the final result:

6.2 Outline effect

The first witcher sense outlines interesting objects in the following way:

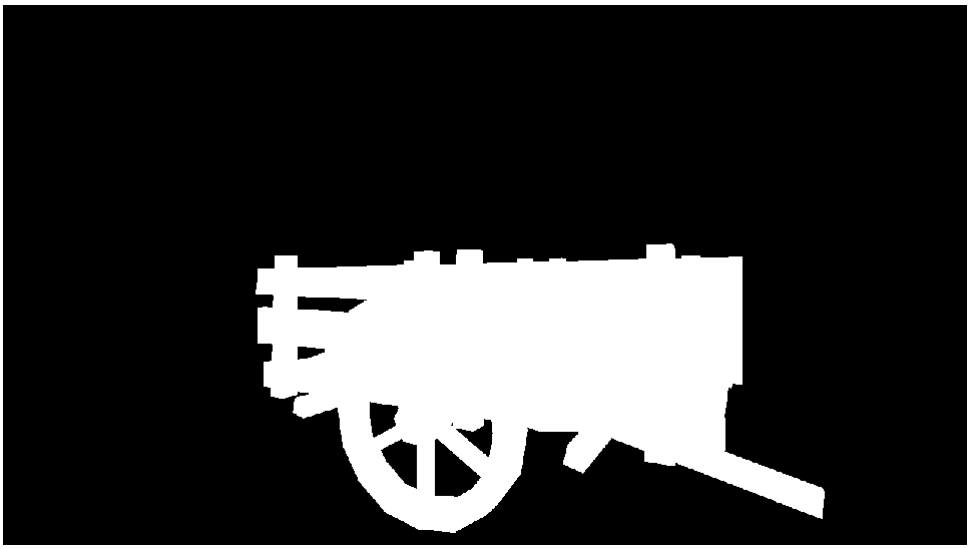
Note that the outline is along the object and the intersection with the player, and it's also half inside and half outside the object. To achieve this effect, we'll need to add another frame buffer with a second texture that we will use to render the border on top of the baseline scene texture.
For example, let's look at this scene:

To create the texture we'll make use of the stencil buffer, that lets us add and remove color information to it. In the first step, we disable the rendering to the texture and enable the writing to the stencil buffer:

```
glColorMask(false, false, false, false);
glDepthMask(false);

glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
```

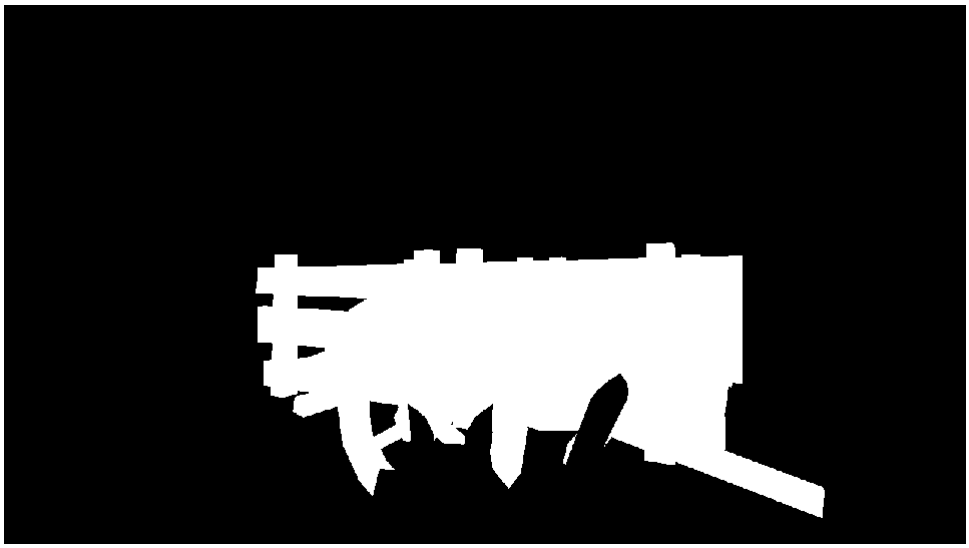
Then, we draw the object, achieving this result on the stencil buffer:



After drawing the object, I remove from the stencil buffer the player that stands in front of it by changing the stencil buffer operation:

```
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
drawPlayer();
```

And this is the stencil buffer after that operation:



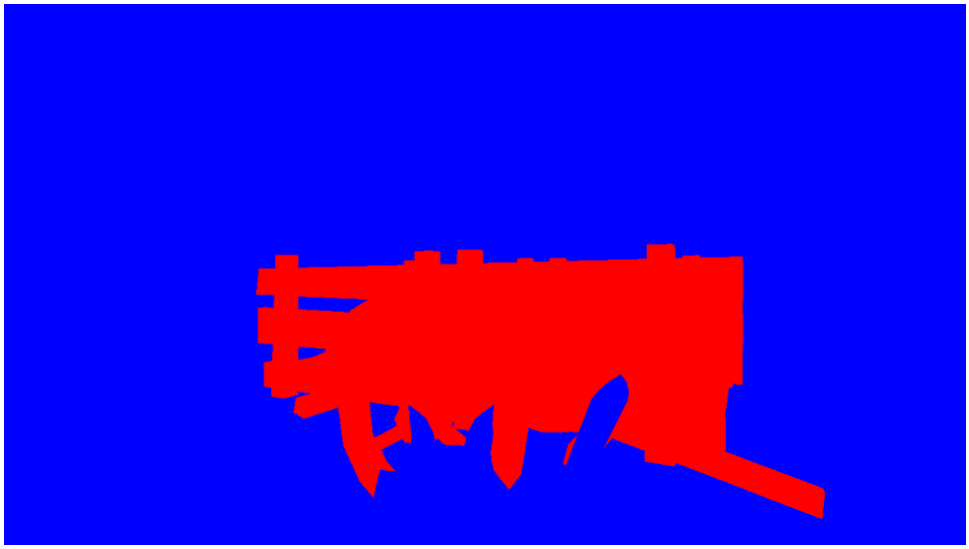
Now that we have the information on the stencil buffer, we re-enable the writing to the render buffer, disable the writing to the stencil buffer and set the stencil operation to draw only where the value in the stencil buffer is 1.

```
glColorMask(true, true, true, true);
glDepthMask(true);

glStencilMask(0x00);

glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilFunc(GL_EQUAL, 1, 0xFF);
```

With this setting, we write on a blue texture using the red color, achieving this result in the frame buffer texture:



With this texture, we can now draw the border in the vertex shader using this algorithm, where I keep only the fragments in the overlapping border:

```
neighbors[] = getNeighbourTexels(distance N);
current = getColor();
if(current.isBlue && neighbors.areBlue) discard;
if(current.isRed && neighbors.areRed) discard;
neighbors[] = getNeighbourTexels(distance N / 5);
alpha = 1.0f;
if(current.isBlue && neighbors.areBlue) alpha = noise();
if(current.isRed && neighbors.areRed) alpha = noise();
return (red, alpha)
```

And this is the final result:

To achieve this effect, we have a total of 3 render passes:

EID	Name
	Frame #403
0	Capture Start
1-85	Colour Pass #1 (1 Targets + Depth)
86-145	Colour Pass #2 (1 Targets + Depth)
146-168	Colour Pass #3 (1 Targets + Depth)
170	SwapBuffers(**Backbuffer Color**)

6.3 Odor effect

The second witcher sense creates an odor scent that can be followed:

To achieve this effect, we first read from the map csv the list of key points where the trail should pass, then we use them to create a Kochanek–Bartels spline. We randomize the starting and ending tangents to create the curve in the XZ plane, and we apply a simple algorithm to have the Y value being 0 at the beginning and at the end, while randomly changing in the middle:

```
tangents = createTangents()
heights = createHeights()
foreach(footprint point) {
    for(i = 0..10) {
        step = i / 10;
        point = vec3(p0 + m0 + m1 + p1, height);
```

```
    }
}
```

Once we have the list of points, we can draw them by calling:

```
glDrawArrays(GL_POINTS, 0, points.size());
```

We have now drawn the points, and we have to add a step to transform them in particles: we will use the Geometry shader for that. Since we would waste computation by using a passthrough geometry shader when it's not needed, we'll create a new shader program to be used for rendering this part.
The geometry shader receives in input the points and tranforms each one into a quad with the normal directed to the camera:

```
void main() {
    gl_Position = gl_in[0].gl_Position + vec4(0, 0, 0, 0);
    tex_coord = vec2(0, 0);
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + vec4(0, 1, 0, 0);
    tex_coord = vec2(0, 1);
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + vec4(0.5625, 0, 0, 0);
    tex_coord = vec2(1, 0);
    EmitVertex();
    gl_Position = gl_in[0].gl_Position + vec4(0.5625, 1, 0, 0);
    tex_coord = vec2(1, 1);
    EmitVertex();
    EndPrimitive();
}
```

The tex_coord variable that contains the UV of the freshly created quad is then passed to the fragment shader to draw the image from the given texture. This is the final result:

6.4 Footprints effect

The third sense is similar to both the previous, meaning that a footprint path appears and it's meant to be followed to a destination:

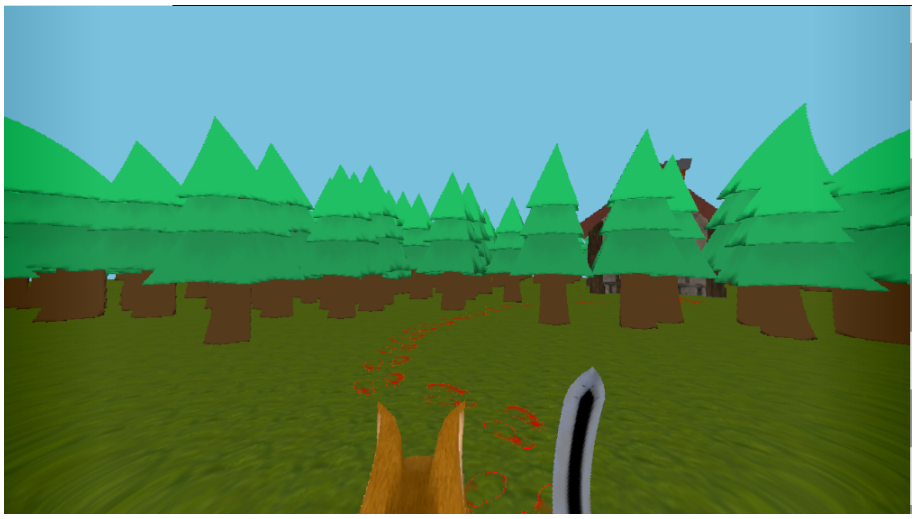
To achieve this effect we start with the same approach as per the odor trail by reading the key points in the map csv. But since we now need the footprints to be approximately at the same distance from each other, we calculate a very high number of intermediate points (100 for each segment) and then pick a subset of points at the same distance:

```
tangents = createTangents()
foreach(footprint point) {
    for(i = 0..100) {
        step = i / 100;
        point = vec3(p0 + m0 + m1 + p1, height);
    }
}
foreach(created points) {
    if(distance(thisPoint, prevPoint) > N) {
        pickPoint();
    }
}
```

Once we've defined the points, we need to rotate the footprint object (which is a plane) to orient it towards the next one:

```
double bearing(double a1, double a2, double b1, double b2) {
    const double TWOPI = 6.2831853071795865;
    const double RAD2DEG = 57.2957795130823209;
    double theta = atan2(b1 - a1, a2 - b2);
    double deg = RAD2DEG * theta * -1;
    return deg;
}
```

The result of this method is the rotation that we apply to the object, and this is the result:



7 Performance

Even with three render passes and the geometry shader, the frame rate is above 60fps in every state of the game and the only bottleneck that caused very low fps was observed before the implementation of the UBO for the trees.
Other performance optimizations were applied, such as:

- Precalculating every static object attributes during the scene load
- Delaying all the calculation until strictly needed with ifs and early returns
- Switching the shader program to avoid passing through the geometry shader when not needed
- Applying 3 different collision detection phases to the camera/objects checks
- Avoiding application of effects and the usage of the additional stencil buffer when not needed