GTA V - Graphics Study

Posted by Adrian Courrèges Nov 2nd, 2015

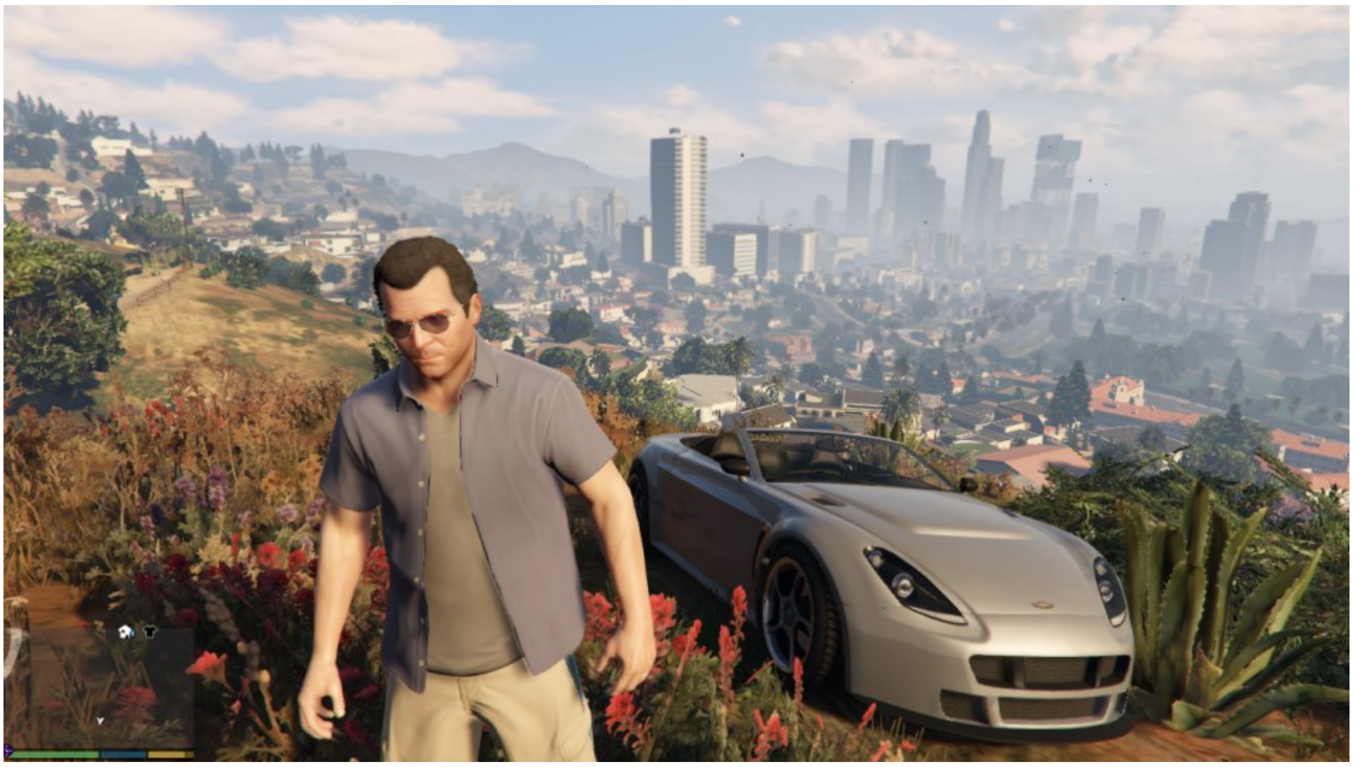
The *Grand Theft Auto* series has come a long way since the first opus came out back in 1997. About 2 years ago, Rockstar released GTA V. The game was an instant success, selling 11 million units over the first 24 hours and instantly smashing 7 world records.

Having played it on PS3 I was quite impressed by the level of polish and the technical quality of the game.
Nothing kills immersion more than a loading screen: in GTA V you can play for hours, drive hundreds of kilometers into a huge open-world without a single interruption. Considering the heavy streaming of assets going on and the specs of the PS3 (256MB RAM and 256MB of video memory) I'm amazed the game doesn't crash after 20 minutes, it's a real technical prowess.

Here I will be talking about the PC version in DirectX 11 mode, which eats up several GBs of memory from both the RAM and the GPU. Even if my observations are PC-specific, I believe many can apply to the PS4 and to a certain extent the PS3.

Dissecting a Frame

So here is the frame we'll examine: Michael, in front of his fancy *Rapid GT*, the beautiful city of *Los Santos* in the background.



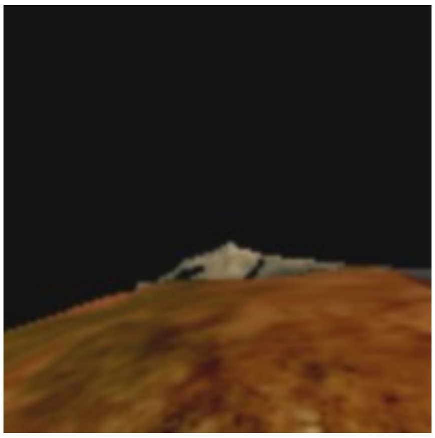
GTA V uses a deferred rendering pipeline, working with many HDR buffers. These buffers can't be displayed correctly as-is on a monitor, so I post-processed them with a simple Reinhard operator to bring them back to 8-bit per channel.

Environment Cubemap

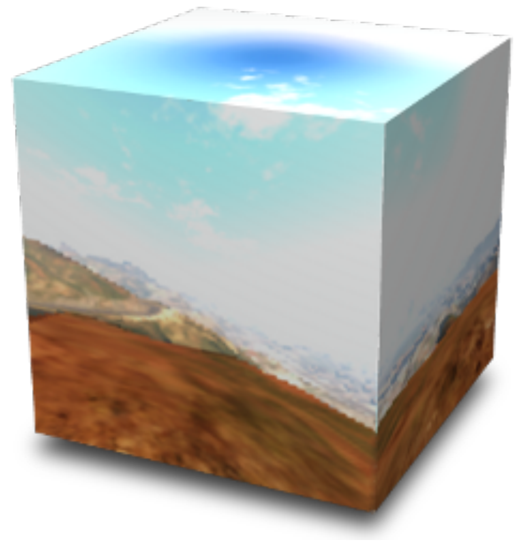
As a first step, the game renders a cubemap of the environment. This cubemap is generated in realtime at each frame, its purpose is to help render realistic reflections later. This part is forward-rendered.
How is such cubemap rendered? For those not familiar with the technique, this is just like you would do in the real world when taking a panoramic picture: put the camera on a tripod, imagine you're standing right in the middle of a big cube and shoot at the 6 faces of the cube, one by one, rotating by 90° each time.

This is exactly how the game does: each face is rendered into a 128x128

HDR texture. The first face is rendered like this:



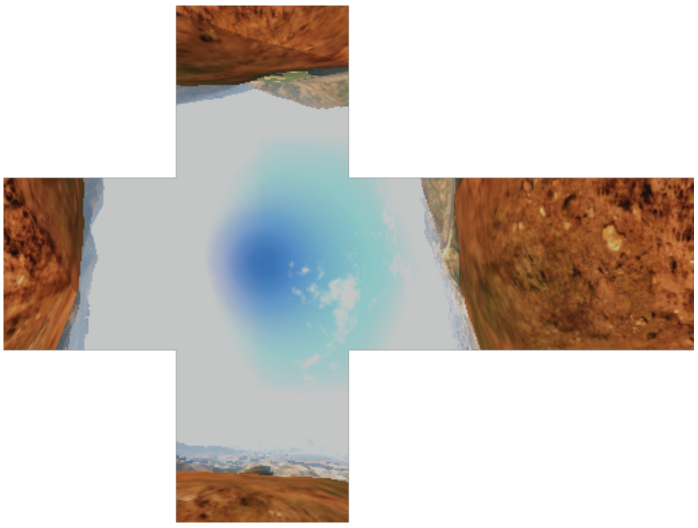
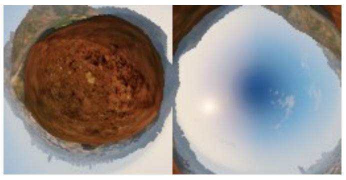
The same process is repeated for the 5 remaining faces, and we finally obtain the cubemap:

Cube seen from the outside	*View from inside the cube*
	*(Drag to change view direction)*

Each face is rendered with about 30 draw calls, the meshes are very low-poly only the “scenery” is drawn (terrain, sky, certain buildings), characters and cars are not rendered.
This is why in the game your car reflects the environment quite well, but other cars are not reflected, neither are characters.

Cubemap to Dual-Paraboloid Map

The environment cubemap we obtained is then converted to a dual-paraboloid map.
The cube is just projected into a different space, the projection looks similar to sphere-mapping but with 2 “hemispheres”.

		*Dual-Paraboloid Map* 

Why such a conversion? I guess it is (as always) about optimization: with a cubemap the fragment shader can potentially access 6 faces of 128x128 texels, here the dual-paraboloid map brings it down to 2 “hemispheres” of 128x128. Even better: since the camera is most of the time on the top of the car, most of the accesses will be done to the top hemisphere.
The dual-paraboloid projection preserves the details of the reflection right at the top and the bottom, at the expense of the sides. For GTA it’s fine: the car roofs and the hoods are usually facing up, they mainly need the reflection from the top to look good.

Plus, cubemaps can be problematic around their edges: if each face is mip-mapped independently some seams will be noticeable around the borders, and GPUs of older generation don’t support filtering across faces. A dual-paraboloid map does not suffer from such issues, it can be mip-mapped easily without creating seams.

Update: as pointed-out in the comments below, it seems GTA IV was also relying on dual-paraboloid map, except it was not performed as a post-process from a cubemap: the meshes were distorted directly by a vertex-shader.

Culling and Level of Detail

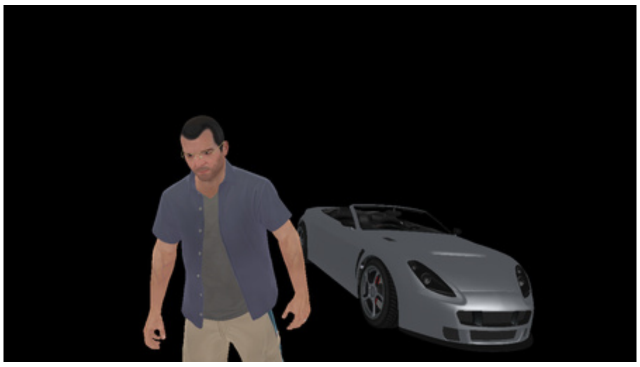
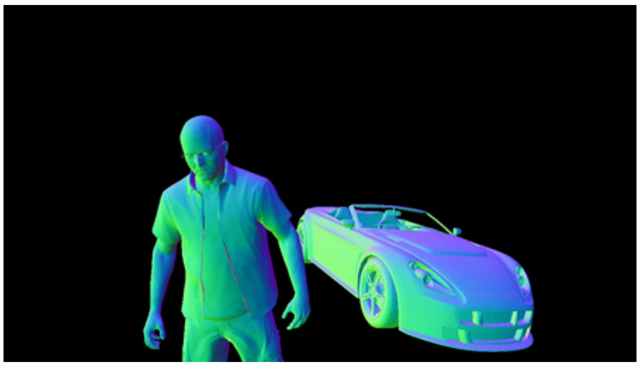
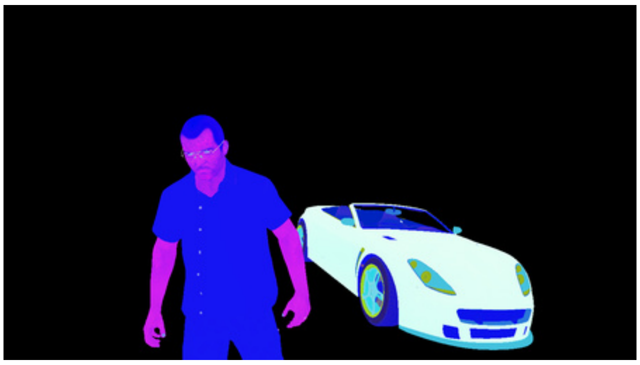
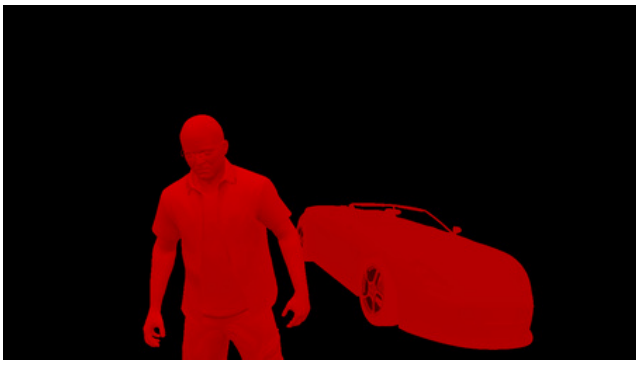
This step is processed by a compute shader, so I don’t have any illustration for it.
Depending on its distance from the camera, an object will be drawn with a lower or higher-poly mesh, or not drawn at all.
For example, beyond a certain distance the grass or the flowers are never rendered. So this step calculates for each object if it will be rendered and at which LOD.

This is actually where the pipeline differs between a PS3 (lacking compute shader support) and a PC or a PS4: on the PS3 all these calculations would have to be run on the Cell or the SPUs.

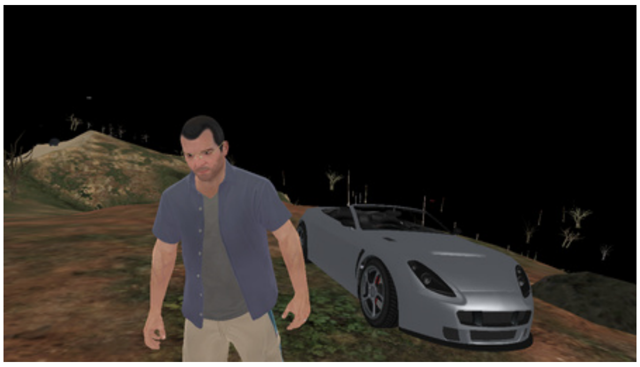
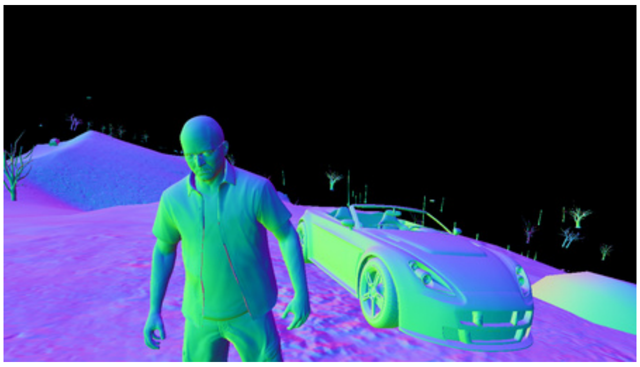
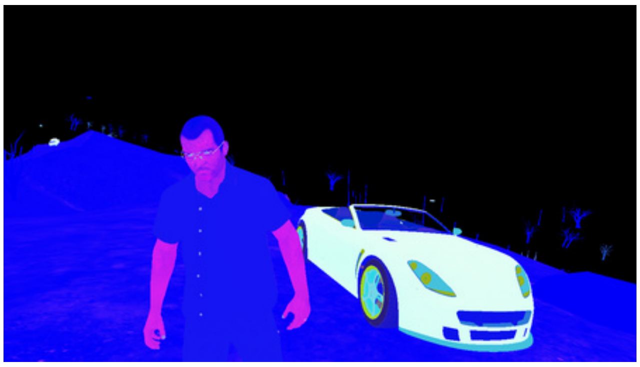
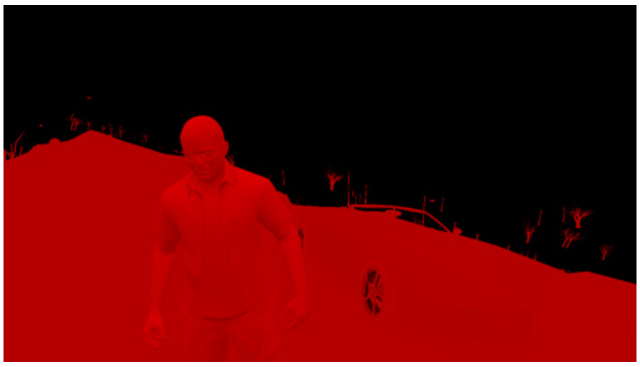
G-Buffer Generation

The “main” rendering is happening here. All the visible meshes are drawn one-by-one, but instead of calculating the shading immediately, the draw calls simply output some shading-related information into different buffers called *G-Buffer*. GTA V uses MRT so each draw call can output to 5 render targets at once.
Later, all these buffers are combined to calculate the final shading of each pixel. Hence the name “deferred” in opposition to “forward” for which each draw call calculates the final shading value of a pixel.
For this step, only the opaque objects are drawn, transparent meshes like glass need special treatment in a deferred pipeline and will be treated later.

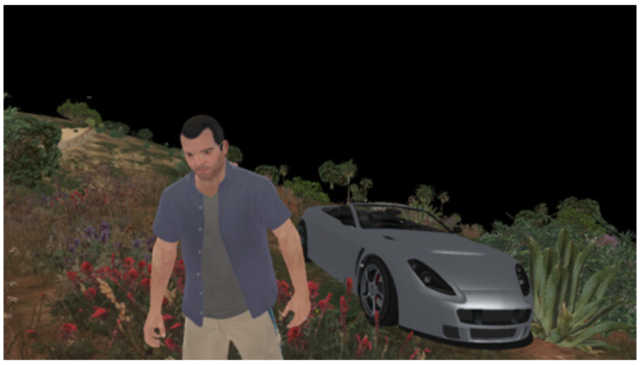
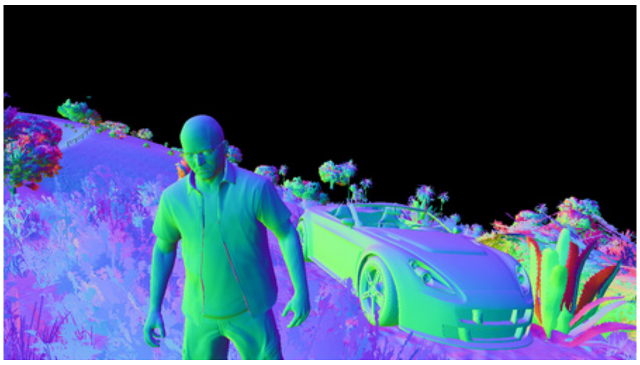
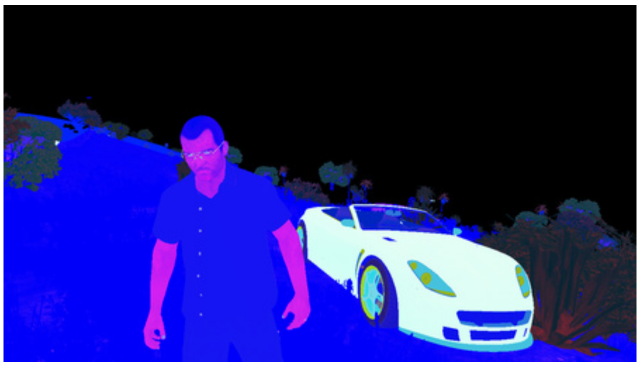
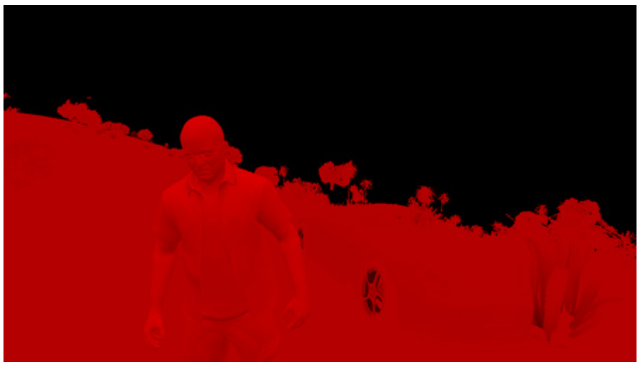
G-Buffer Generation: 15%

 **Diffuse**	 **Normal**
 **Specular**	 **Irradiance**

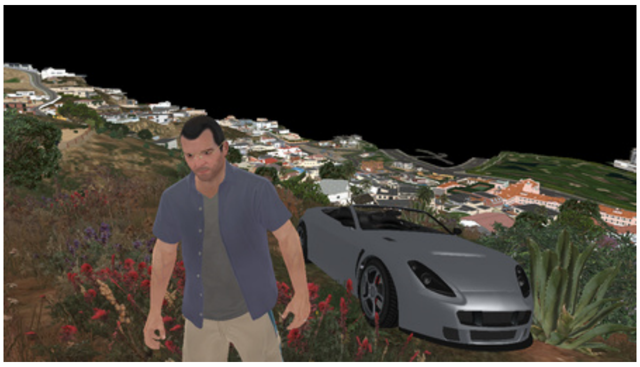
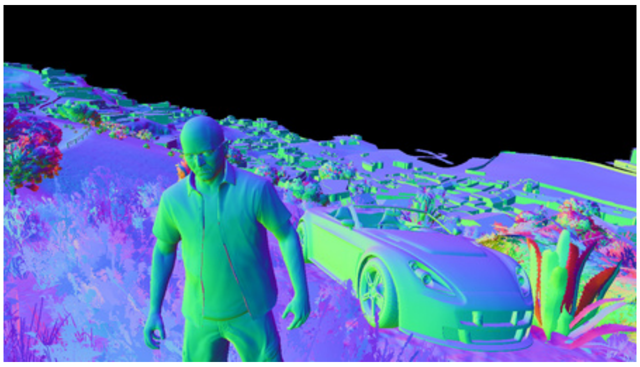
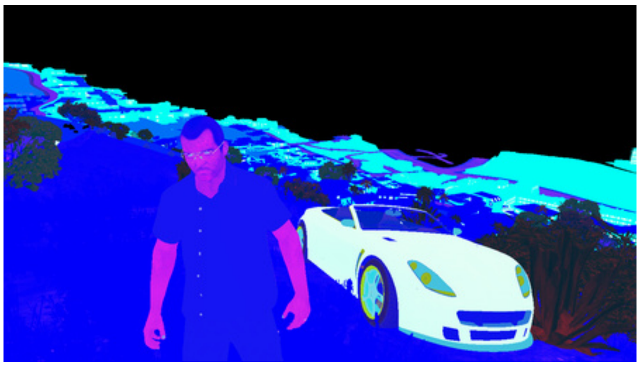
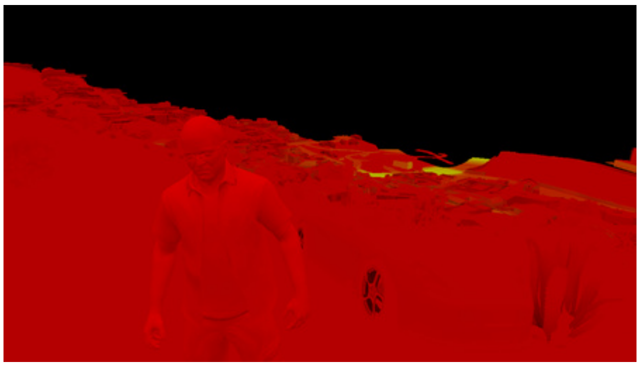
G-Buffer Generation: 30%

 Diffuse	 Normal
 Specular	 Irradiance

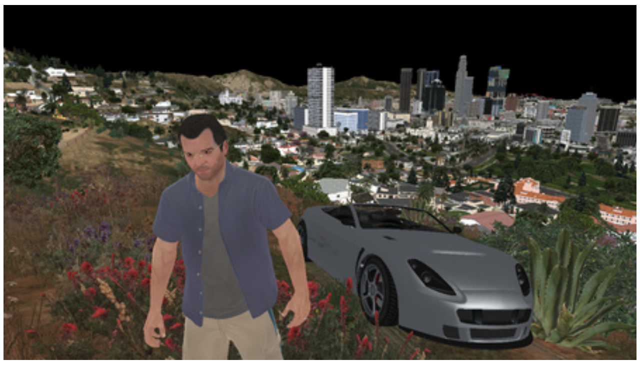
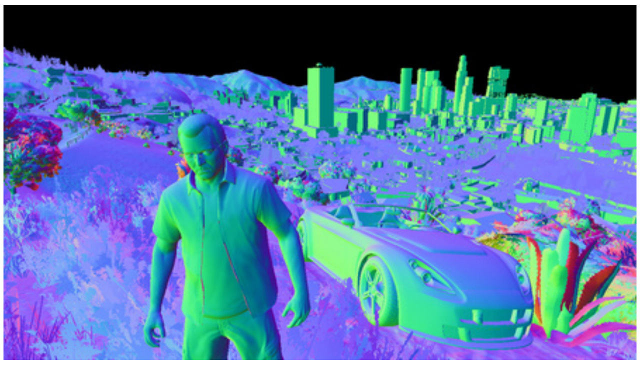
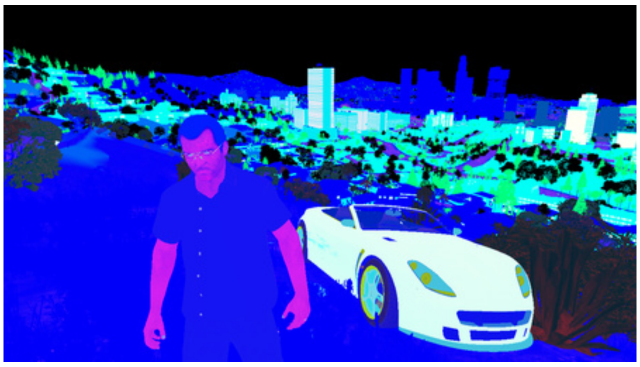
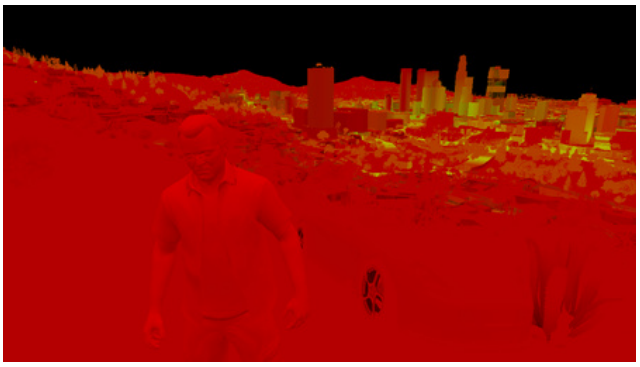
G-Buffer Generation: 50%

 Diffuse	 Normal
 Specular	 Irradiance

G-Buffer Generation: 75%

 **Diffuse**	 **Normal**
 **Specular**	 **Irradiance**

G-Buffer Generation: 100%

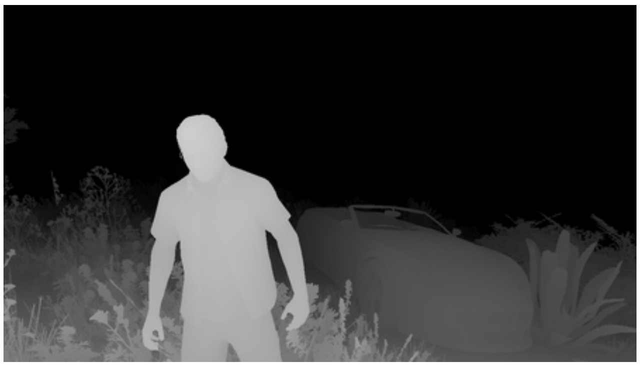
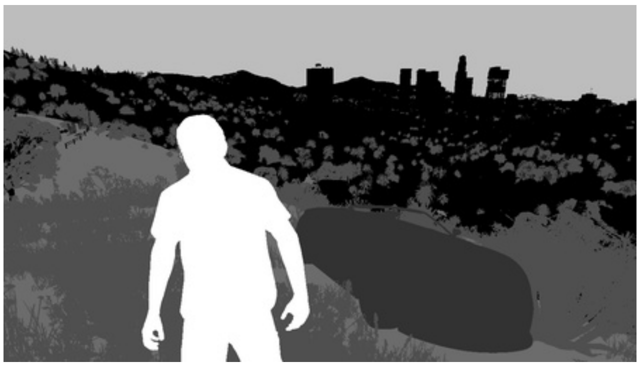
 **Diffuse**	 **Normal**
 **Specular**	 **Irradiance**

All these render targets are LDR buffers (RGBA, 8 bits per channel) storing different information involved later in the calculation of the final shading

value.
The buffers are:

- **Diffuse map**: it stores the “intrinsic color” of the mesh. It represents a property of the material, it is in theory not influenced by the lighting. But do you notice the white highlights on the car’s hood? Interestingly GTA V calculates the shading resulting from the sun directional light before outputting to the diffuse map.
  The alpha channel contains special “blending” information (more on that later).
- **Normal map**: it stores the normal vector for each pixel (R, G, B). The alpha channel is also used although I am not certain in which way: it looks like a binary mask for certain plants close to the camera.
- **Specular map**: it contains information related to specular/reflections:
  - Red: specular intensity
  - Green: glossiness (smoothness)
  - Blue: fresnel intensity (usually constant for all the pixels belonging to the same material)
- **Irradiance map**: the Red channel seems to contain the irradiance each pixel receives from the sun (based on the pixel’s normal and position, and the sun shining direction). I am not 100% sure about the Green channel, but it looks like the irradiance from a second light source. Blue is the emissive property of the pixel (non-zero for neon, light bulbs). Most of the alpha channel is not used except for marking the pixels corresponding to the character’s skin or the vegetation.

So, I was talking before about outputting simultaneously to 5 render targets, but I only presented 4 of them.
The last render target is a special *depth-stencil buffer*. Here is what it looks like at the end of this pass:

 **Depth**	 **Stencil**

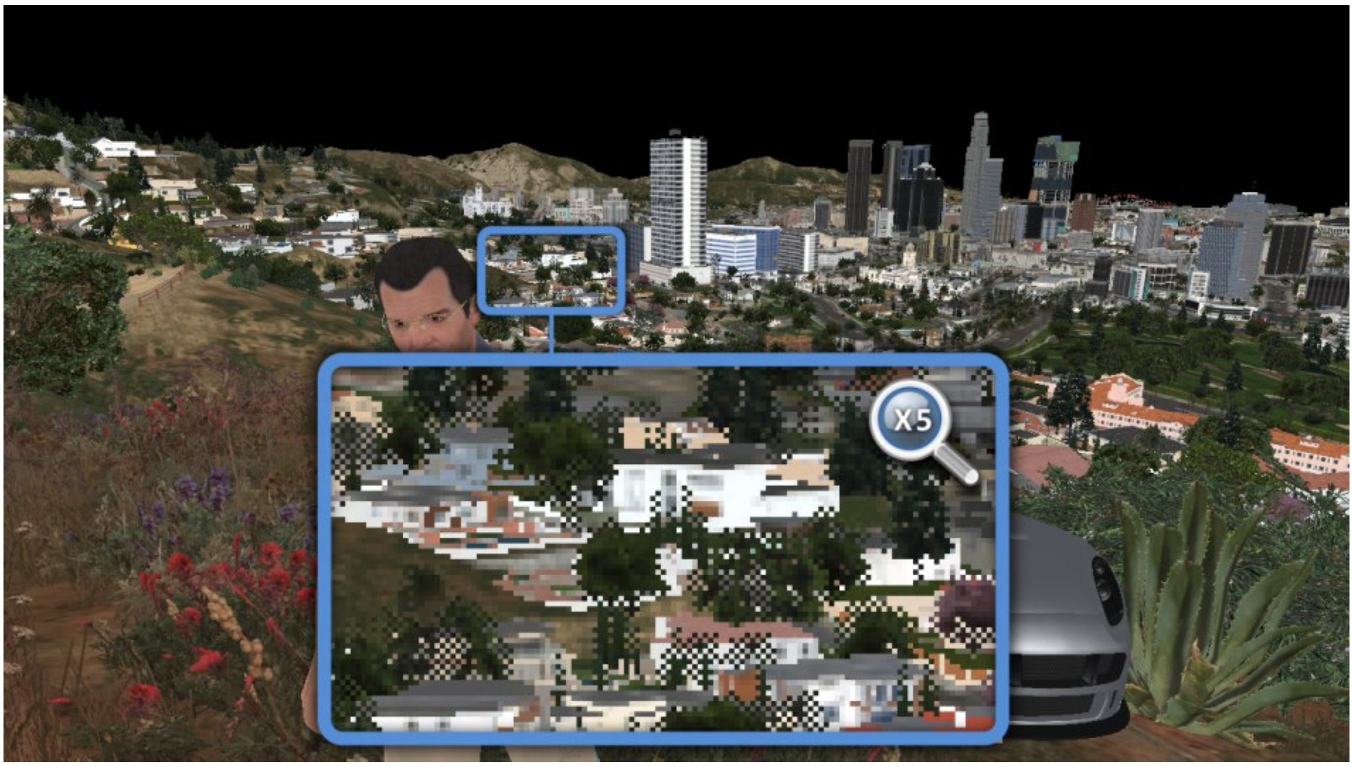
- **Depth map**: it stores the distance of each pixel from the camera. Intuitively you would expect far pixels to be white (depth of 1) and closer pixels to be darker. This is not the case here: GTA V seems to be using a logarithmic Z-buffer, reversing Z. Why do this? Due to the way they are encoded, floating point numbers have much more precision when closer to 0. Here, reversing Z leads to more precision when storing the depth of very distant objects, hence greatly reduces Z-fighting. Given the long draw distance of the game such trick was necessary. It's nothing new though, *Just Cause 2* for example was using a similar technique.
- **Stencil**: it is used to identify the different meshes drawn, assigning the same ID to all the pixels of a certain category of meshes. For example, some stencil values are:
    - `0x89`: the character controlled by the player
    - `0x82`: the vehicle driven by the player
    - `0x01`: NPCs
    - `0x02`: vehicles like cars, bikes…
    - `0x03`: vegetation and foliage
    - `0x07`: sky

All these buffers were generated in about 1900 draw calls.

Notice that the scene is rendered "front-to-back", this is a way to optimize the rendering thanks to "early Z rejection": as the scene is drawn, more and more fragments fail the depth test because they are occluded by a closer pixel drawn previously. When it is known a pixel will fail the depth test, the GPU can automatically discard it without even executing the pixel shader.

When you have heavy pixel shaders, "back-to-front" rendering (or Painter's algorithm) is the worst choice performance-wise, "front-to-back" on the other hand is optimal.

Small digression now, to explain the role of the alpha channel in the diffuse map. Take a look at the screenshot below:

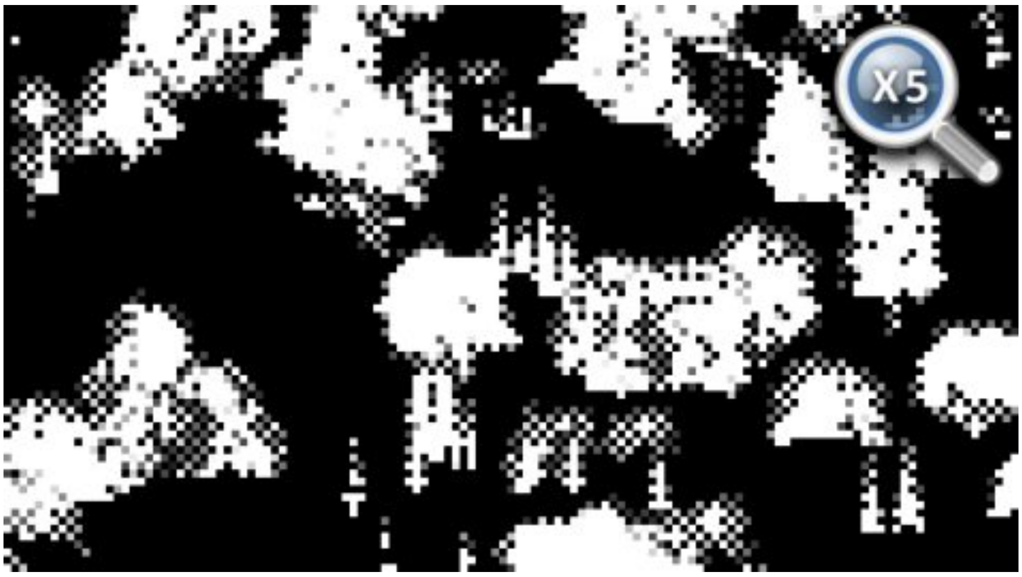


Zoom on the Diffuse Map

Do you notice some pixels missing? It's especially visible for the trees, it's like their sprites lack some texels.
I noticed such artifacts a few times on PS3 and I was puzzled at the time. Could it be aliasing when the texture sprite becomes really tiny? I can see now they're all mip-mapped correctly so it's not this.
This pattern is really specific, like a checkerboard, could it be that… the game skips the rendering of 1 out of 2 pixels?
To make sure, I looked into the D3D bytecode. And sure enough, it was here:

Fragment Shader Assembly

```
dp2 r1.y, v0.xyxx, l(0.5, 0.5, 0.0, 0.0)  // Dot product of the pixel's (x,y) with (0.5, 0.5)
frc r1.y, r1.y                            // Keeps only the fractional part: always 0.0 or 0.5
lt r1.y, r1.y, l(0.5)                     // Test if the fractional part is smaller than 0.5
```



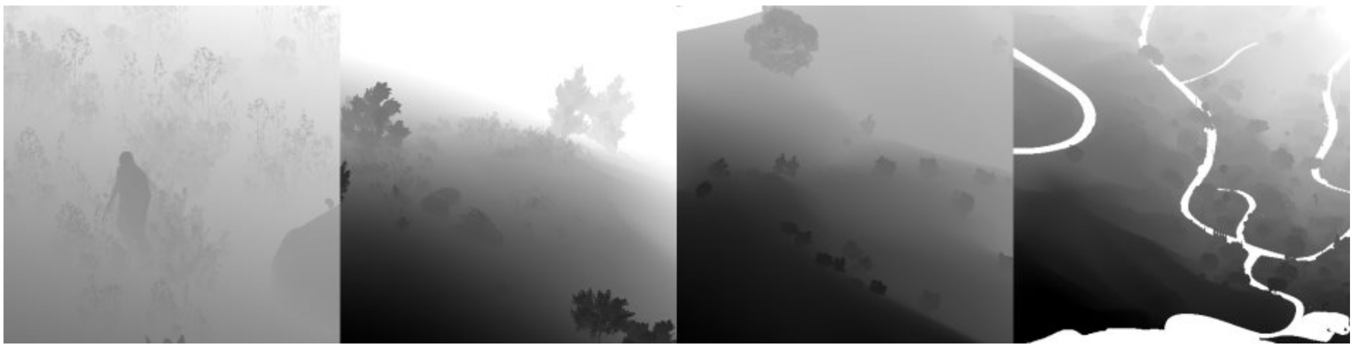
Diffuse's Alpha Channel

All these instructions are simply equivalent to the test `(x + y) % 2 == 0` which 1 out of 2 pixels always pass. (with *x* and *y* being the pixel's coordinate)
This is just one of several conditions leading to discarding a pixel (another one being having an alpha < 0.75) but it is enough to explain the dithering pattern.

To remember which meshes were drawn in such "dithered mode", the information is stored in the alpha channel of the diffuse map, which looks like the picture on the right.
So why are some models drawn like this? Could it be to save on fillrate or shading calculation? Not really because GPUs don't have such granularity: pixels are shaded in square of 2x2, not individually. It's not about performance, it's about LOD transition: this dithering pattern makes opaque meshes look a bit transparent when they transition between LODs. This technique is actually called alpha stippling.

Shadows

The game uses CSM (cascaded shadow maps): 4 shadow maps are generated into a 1024x4096 texture. Each shadow map is created for a different camera frustum, the frustum getting bigger and encompassing a higher portion of the scene as the iteration goes. This ensures the shadows near the player are stored with higher resolutions, while shadows further away have fewer details. Here is an overview of the depth information of the 4 maps:

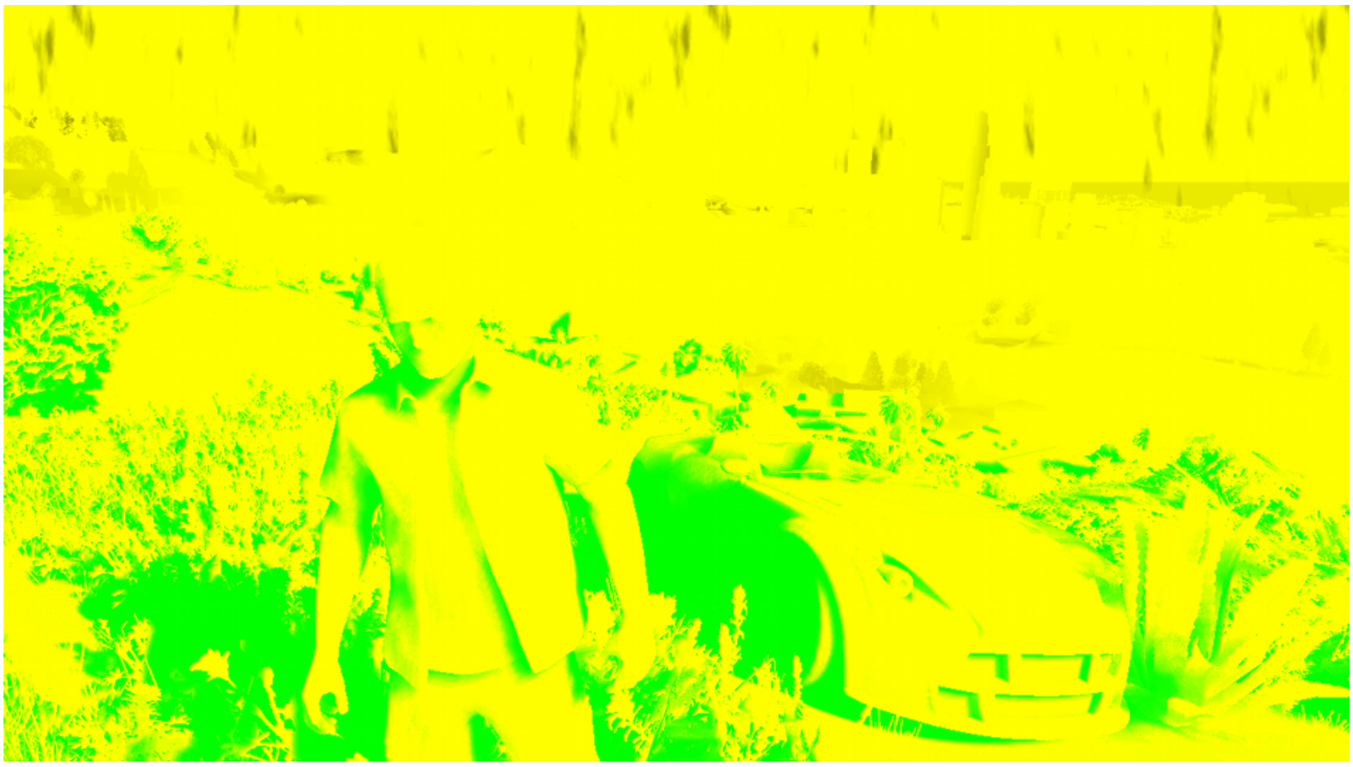


Shadow Maps

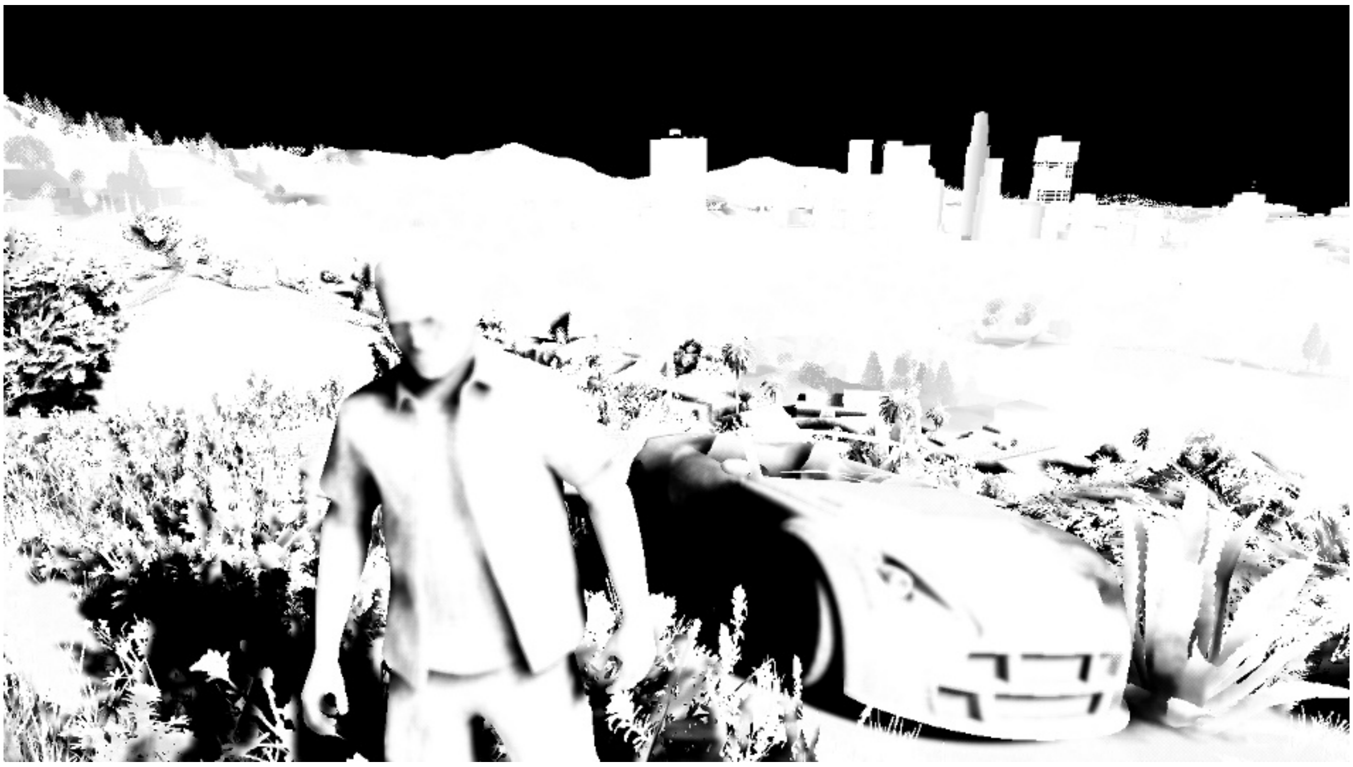
This step can potentially have a high cost since the scene needs to be re-rendered 4 times, but frustum-culling avoids rendering unnecessary polygons. Here the CSM generation is achieved in about 1000 draw calls.

From this depth information, we can calculate the shadow cast on each pixel. The engine stores the shadow information in a render target: shadows cast by the sun directional light are in the Red channel, the ones cast by the clouds in the atmosphere are in both the Red and the Green channels.

The shadow maps are sampled with a dithering pattern (if you look closely at the texture below, the red channel displays some checkerboard-like artifacts), this is to make the shadow borders smoother.
These artifacts are then corrected: sun shadows and cloud shadows are combined into a single buffer, some depth-aware blur is performed and the result is stored into the alpha channel of the specular map.



Sun shadows (green tint)
Cloud shadows (gray tint)



Blurred Shadows

A quick note about the blur operation: it is quite expensive because it needs to do many taps from multiple textures. So to alleviate the load, just before performing the blur, an “early out” texture is created: the shadow buffer is downscaled to 1/8th, and a light-weight blur is performed by a pixel shader making 4 calls to `Gather()`. This can give a rough estimate of which pixels are fully lit. Then when the full depth-aware blur is performed, the first step consists in reading this "early out" buffer: if the current pixel appears to be fully lit the pixel shader immediately outputs 1 and skips all the heavy blur computation.

Planar Reflection Map

I won't go too much into details since reflections are explained more in details in *Part 2* and the effect is barely visible in this scene, but this step generates a reflection map for the ocean surface.

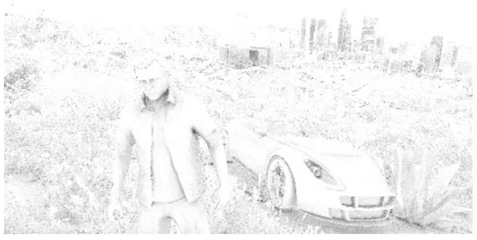
Basically the scene is drawn again (in 650 draw calls) into a tiny 240x120 texture, but "upside-down", to appear like a reflection on a water surface.

Screen Space Ambient Occlusion

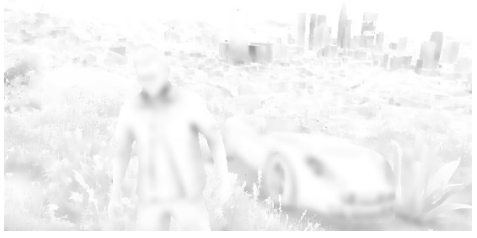
A linear version of the depth-buffer is computed, and then from it the SSAO map is created.

A first noisy version is generated, then a depth-aware blur is applied in 2 consecutive passes (horizontal and vertical) to smooth-out the result.

All the work is done at half the original resolution in order to increase performance.



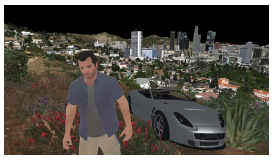
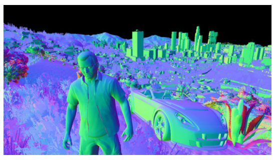
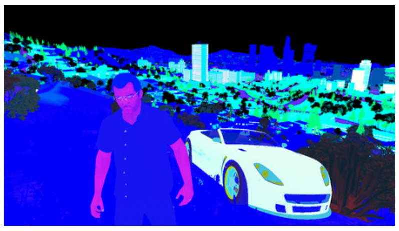
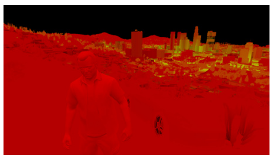
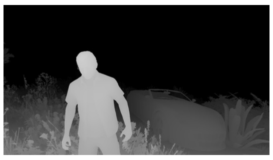
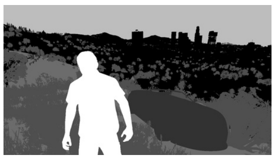
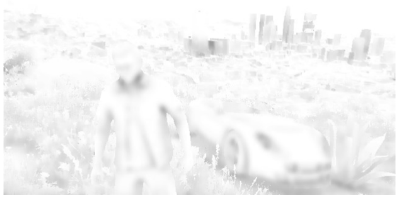
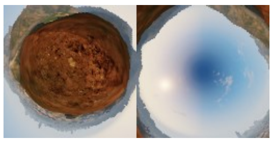
SSAO Map (Noisy)

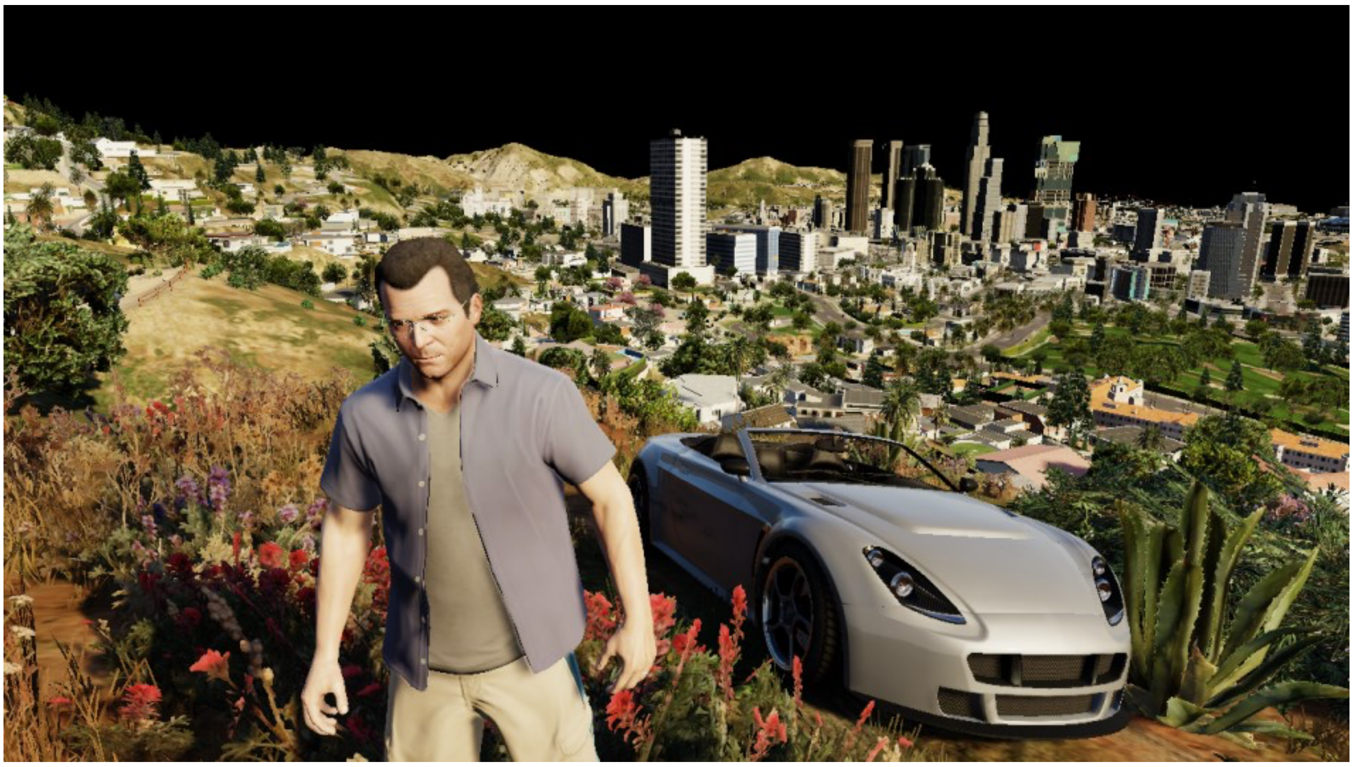


SSAO Map (Blurred)

G-Buffer Combination

Time to finally combine all these buffers which have been generated!
A pixel shader fetches the data from the different buffers and computes the final shading value of the pixel in HDR.
In the case of a night scene, lights and their irradiance would also now be added one by one on the top of the scene.

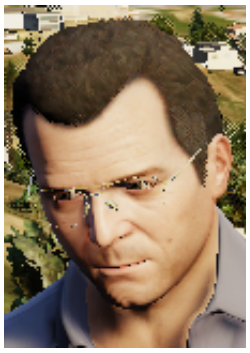
 **Diffuse**	 **Normal**	 **Specular + Shadow**	 **Irradiance**
 **Depth**	 **Stencil**	 **SSAO**	 **Reflection**



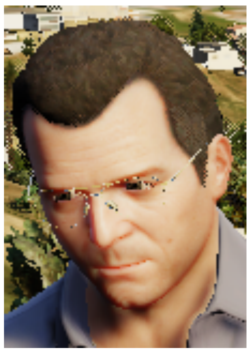
Combination of G-Buffers

It’s starting to take shape nicely, although we’re still missing the ocean, the sky, transparent objects…
But first things first: Michael’s appearance needs to be enhanced.

Subsurface Scattering



SSS (Before)



SSS (After)

The shading of Michael's skin is a bit off: there are very dark areas on his face, like if his body was made of thick plastic instead of flesh.

This is why a pass of SSS is performed, simulating the transport of light within the skin. Look at his ears or his lips: after the SSS pass the light is now bleeding through them, giving a red tint which is exactly what you would expect to happen in the real world.

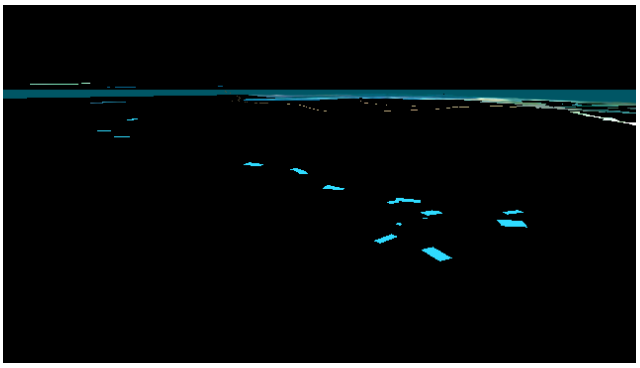
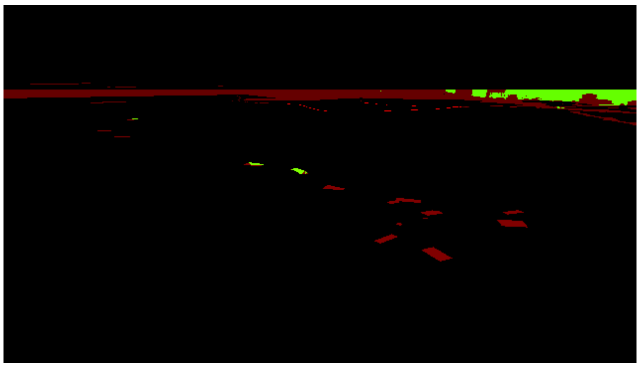
How was the SSS applied to Michael only? First only his silhouette is extracted. This is possible thanks to the stencil buffer generated before: all of Michael's pixels have a value of `0x89`. So we can get Michael's pixels, great, but we want to apply the SSS only to the skin, not to the clothes. Actually, when all the G-Buffers were combined, in addition to the shading data stored in the RGB, some data was being written to the alpha channel too. More precisely, the irradiance map and the specular map alpha channels were used to create a binary mask: the pixels belonging to Michael's skins and to some plants are set to 1 in the alpha channel. Other pixels like the clothes have an alpha of 0.
So the SSS can be applied by providing as input simply the combined G-Buffer target and the depth-stencil buffer.

Now, you might think this amounts to a lot of computation for just a subtle, local improvement. And you would be right, but don't forget that when playing the game, as humans we instinctively tend to look at the character's face a lot – any rendering improvement made to the face can be a big win for the feeling of immersion. In the game SSS is applied to both your character and the NPCs.

Water

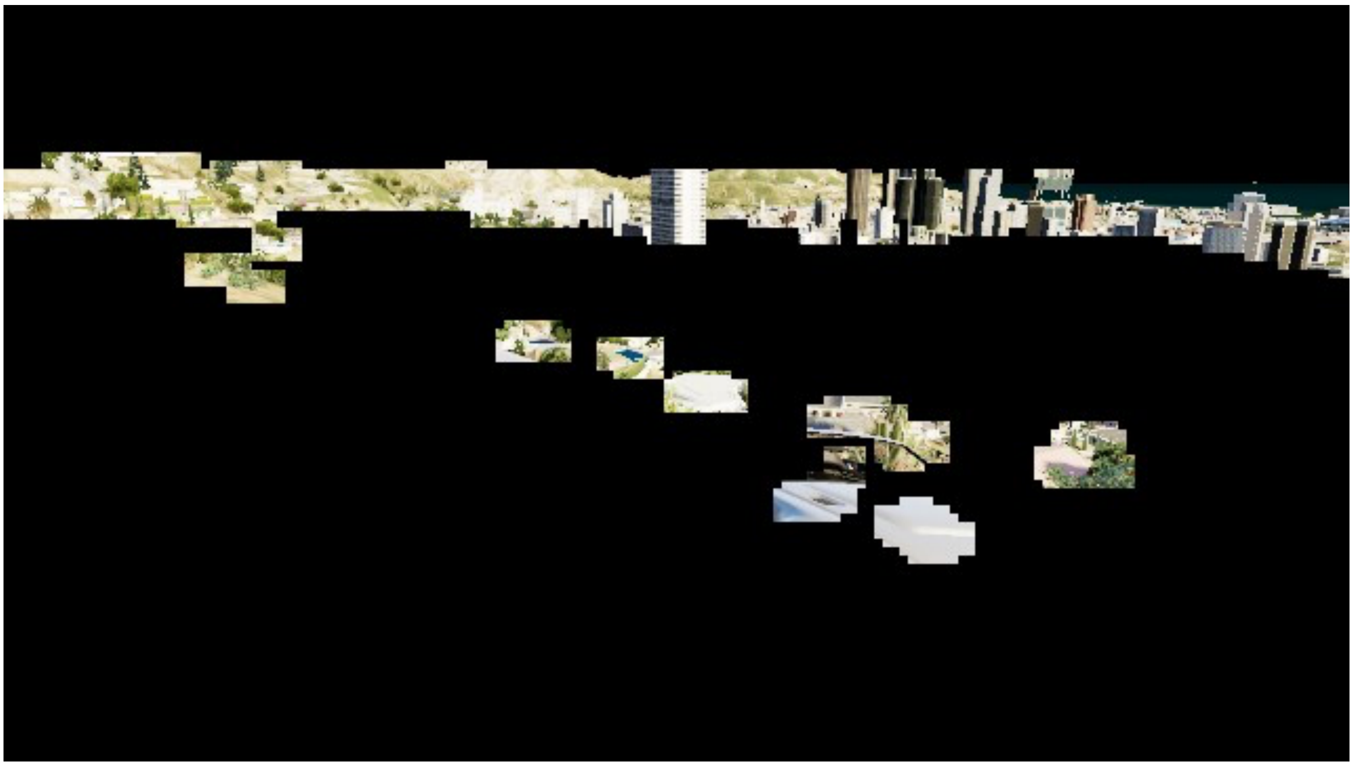
There is not much water in this scene but still, we have the ocean in the back, some swimming pools here and there.
The water rendering in GTA V handles both reflection and refraction.

The logarithmic Z-buffer created previously is used to generate a second version: linear this time, at half the resolution.
The ocean and pools are drawn one by one, in MRT mode, outputting to several targets at once:

 **Water Diffuse**	 **Water Opacity**

- **Water Diffuse map**: it is the intrinsic color of the water.
- **Water Opacity map**: the red channel seems to store some opacity property of the water (for example ocean is always 0.102, pools are always 0.129). The green channel stores how deep a pixel is from the water surface (deep pixels have a more opaque water with a strong contribution from the diffuse map, whereas water for shallow pixels is almost transparent).
  Note that all the pools are rendered unconditionally, even if they end up hidden behind another mesh in the scene, they all appear in the red channel. For the green channel however, only the pixels really visible are calculated, the only “water” pixels that make it into the final image.

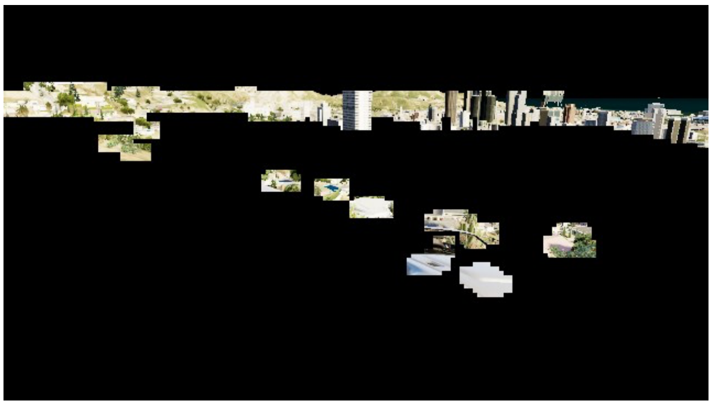
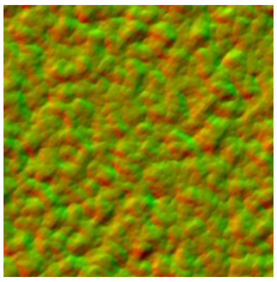
We can now combine the previous buffers we created and generate a refraction map:

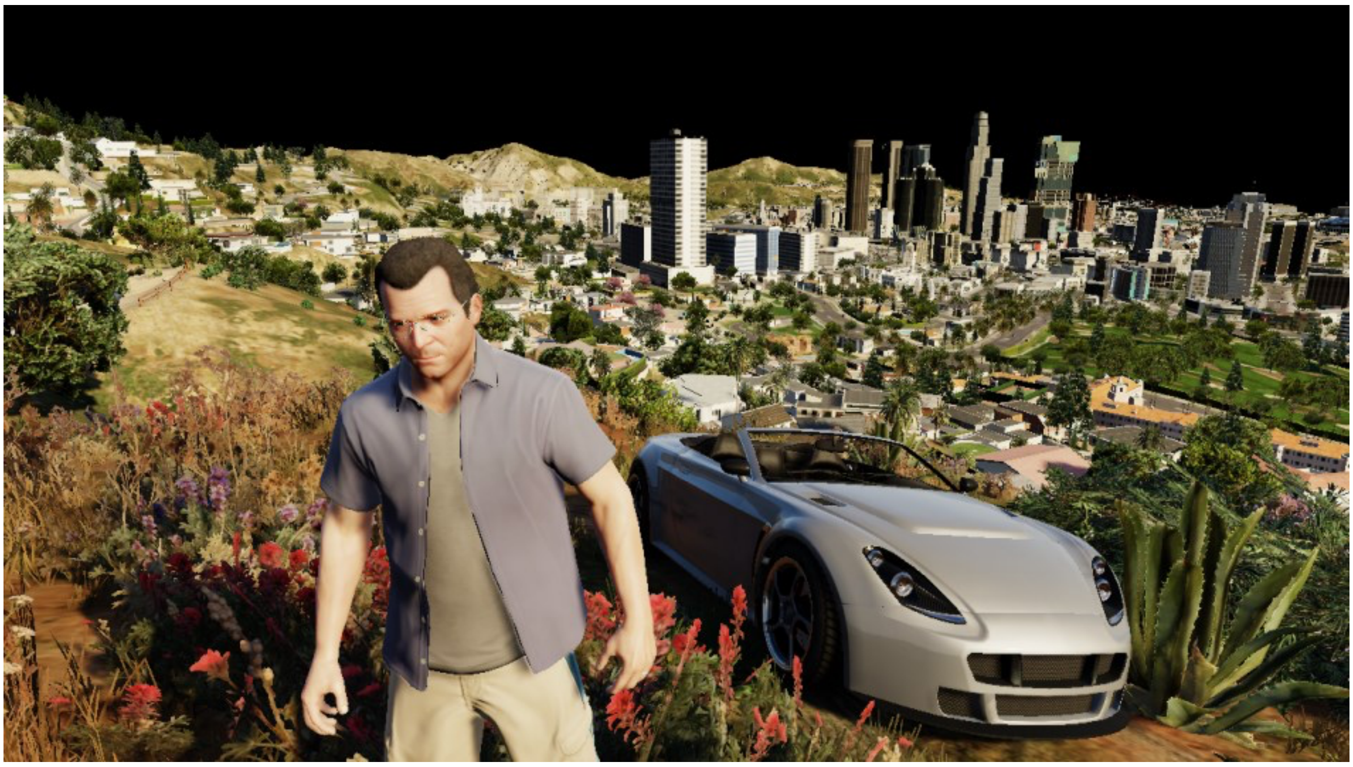


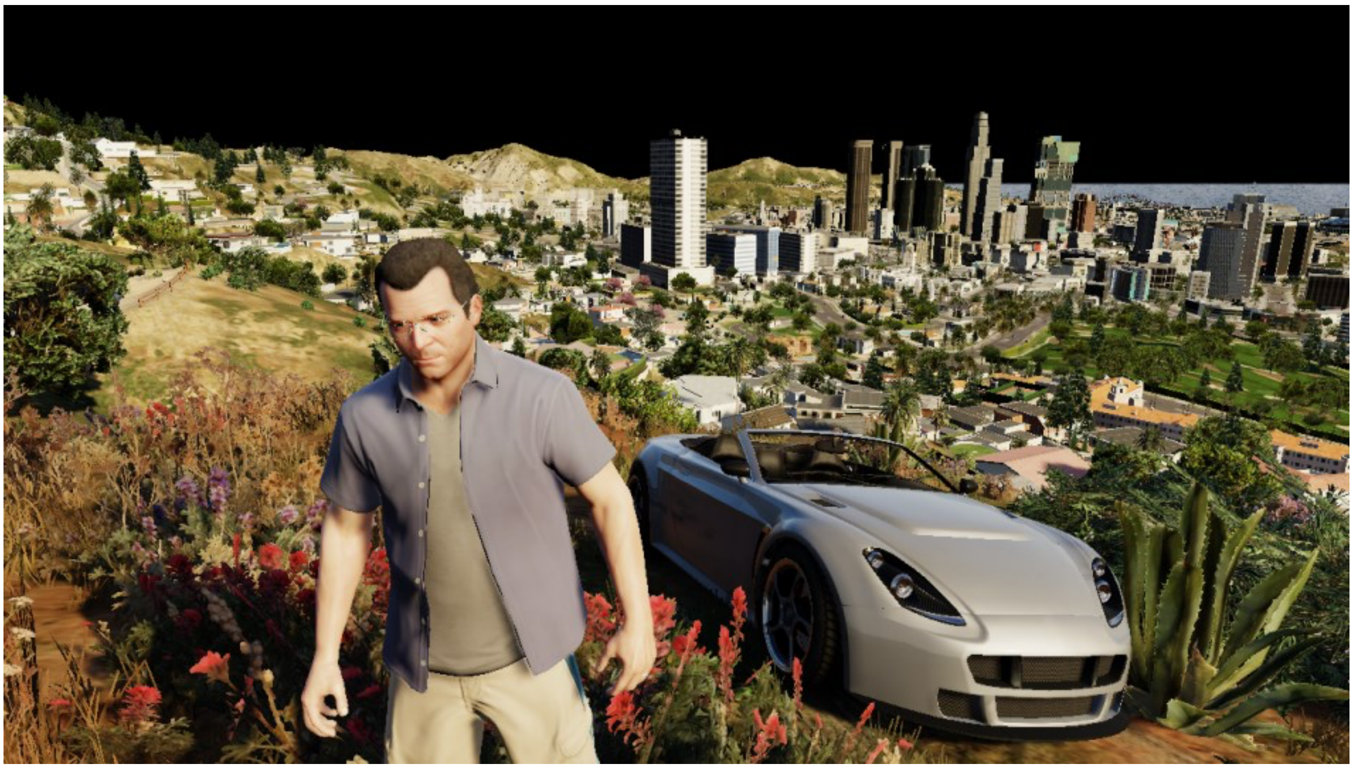
Water Refraction Map

In this refraction map, the pools are filled with water (the deeper the water the bluer), caustics are also added.

We can now proceed to the final rendering of the water: once again all the meshes of the ocean, the pool surfaces are drawn one by one, but this time combining the reflection and the refraction together, with some bump maps to perturb the surface normals.

Planar Reflection	 **Refraction**	 **Bump Maps**


Water (Before)


Water (After)

Atmosphere

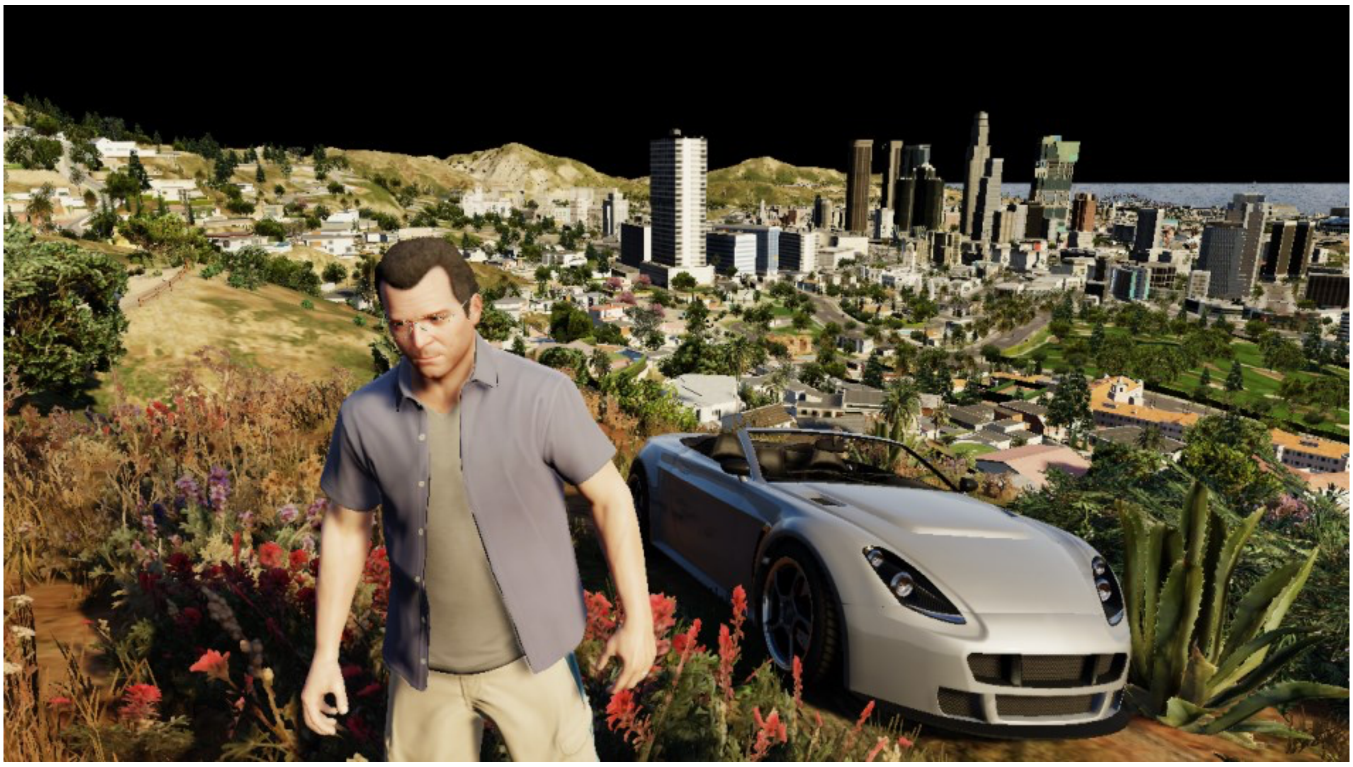
Light Shaft

A light-shaft map – also called “volumetric shadows” – is created: its role is to darken the atmosphere/fog which is not directly lit by the sun.

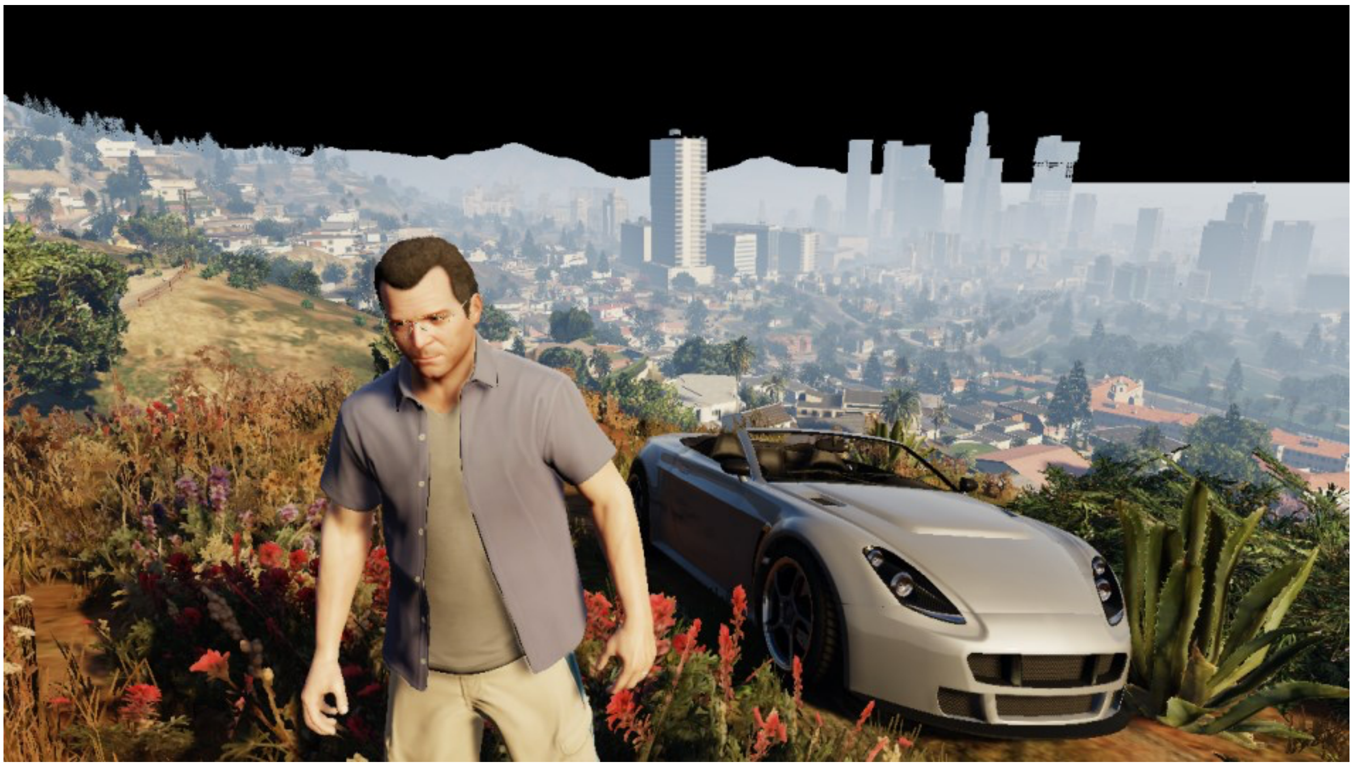
The map is generated at half the resolution, by ray-marching each pixel and evaluating against the sun shadow map.
After a first noisy result is obtained, the buffer is blurred.

The next step consists in adding a fog effect to our scene: it conveniently hides the lack of details of the low-poly buildings in the distance.
This pass reads from the light-shaft map (which has little influence in this shot) and the depth-buffer to output the fog information.

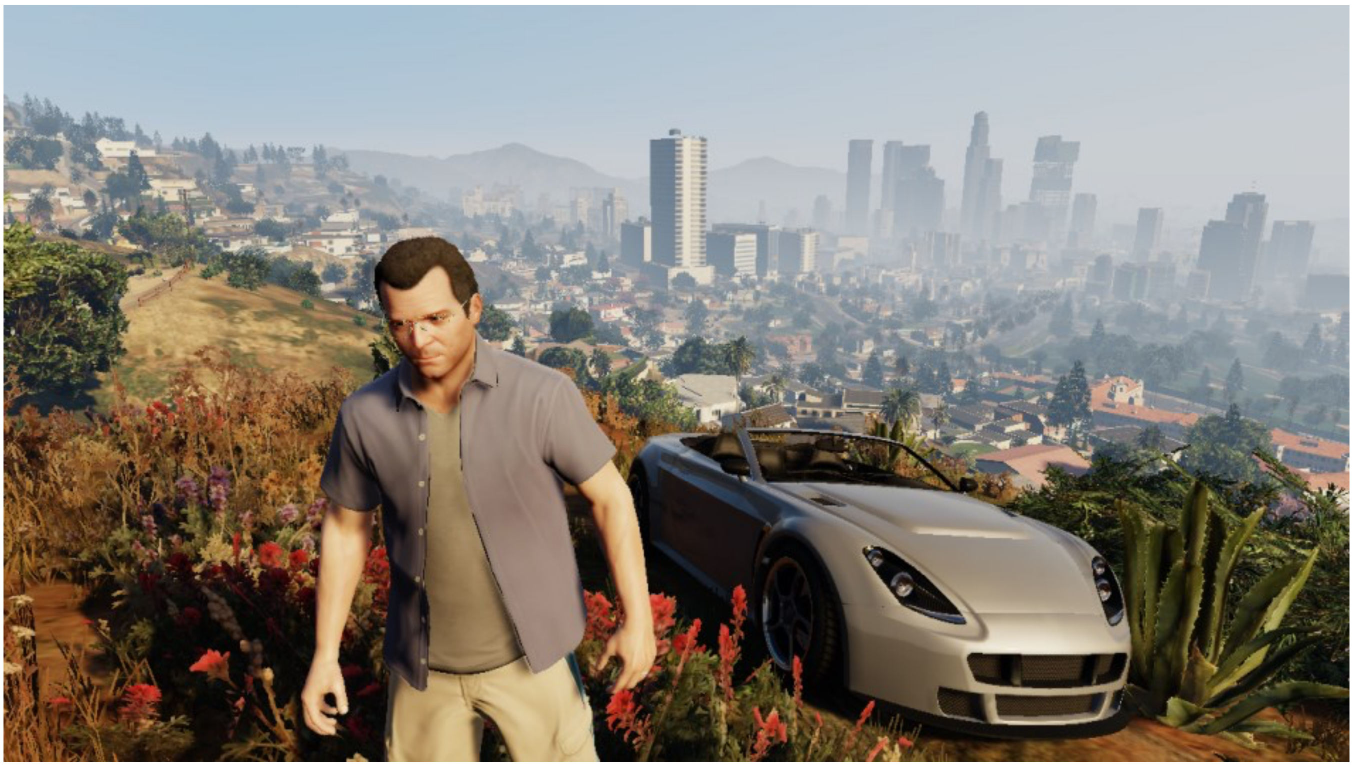
Then the sky is rendered, followed by the clouds.



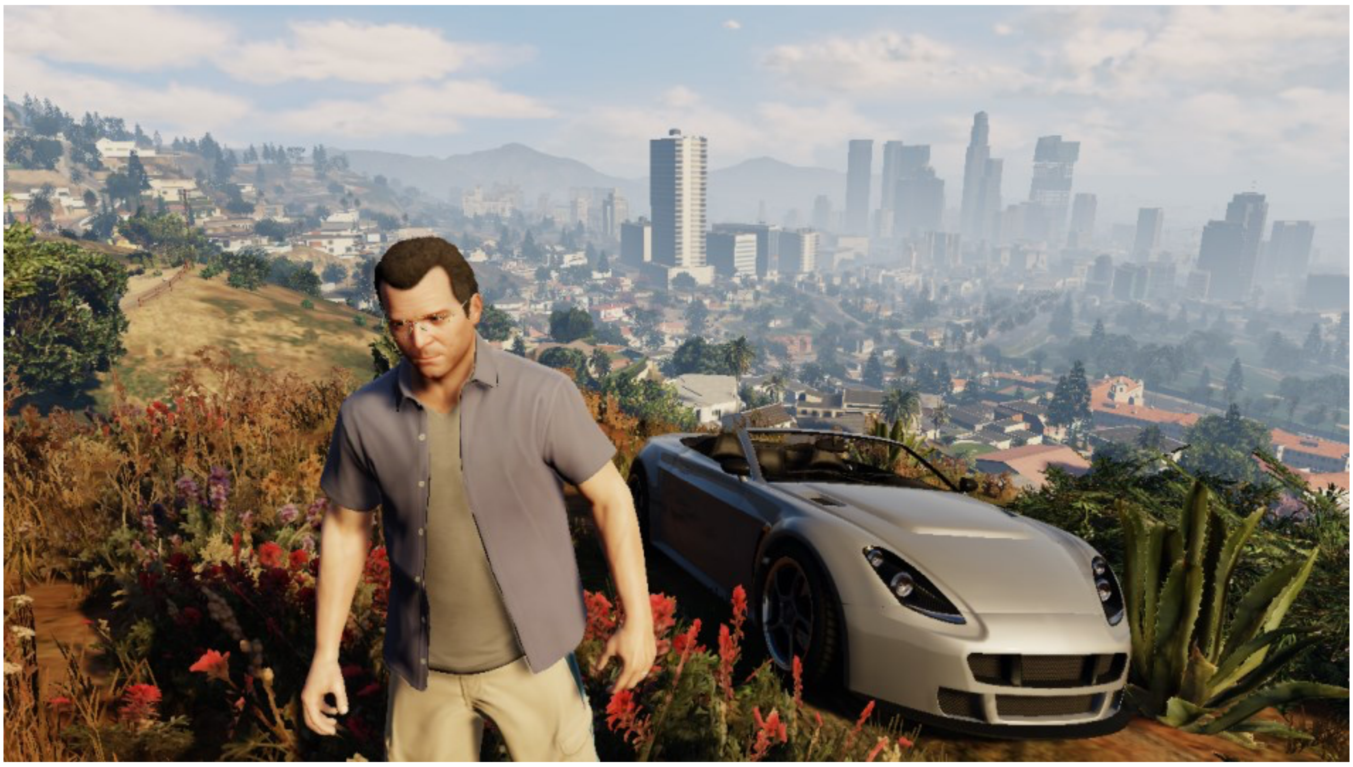
Base



Base + Fog



Base + Fog + Sky

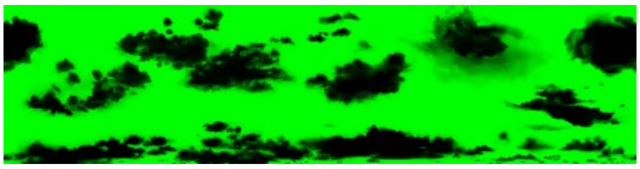
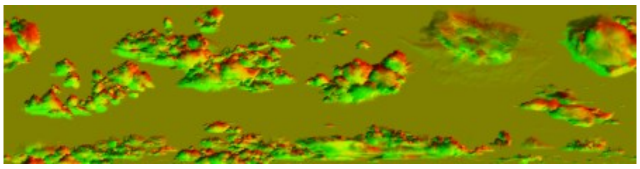


Base + Fog + Sky + Clouds

The sky is actually rendered in a single draw call: the mesh used is a huge dome covering the entire scene. (see on the right)
This step uses as input some textures similar to Perlin noise.

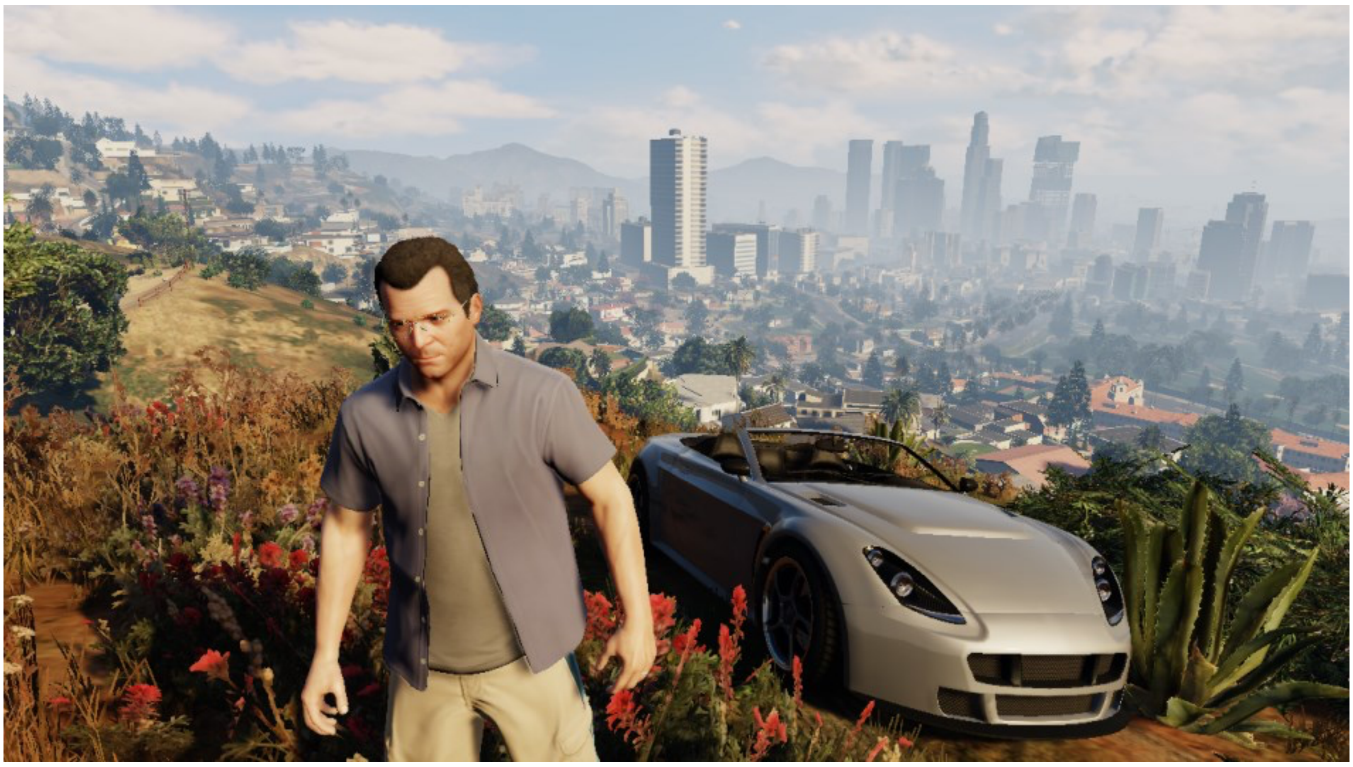
Sky Dome

The clouds are rendered in a similar way: a large mesh, with a shape of ring this time, is rendered in the horizon. One normal map and one density map are used to render the clouds: these are big 2048x512 textures, which are also seamless (they loop on their left and right sides).

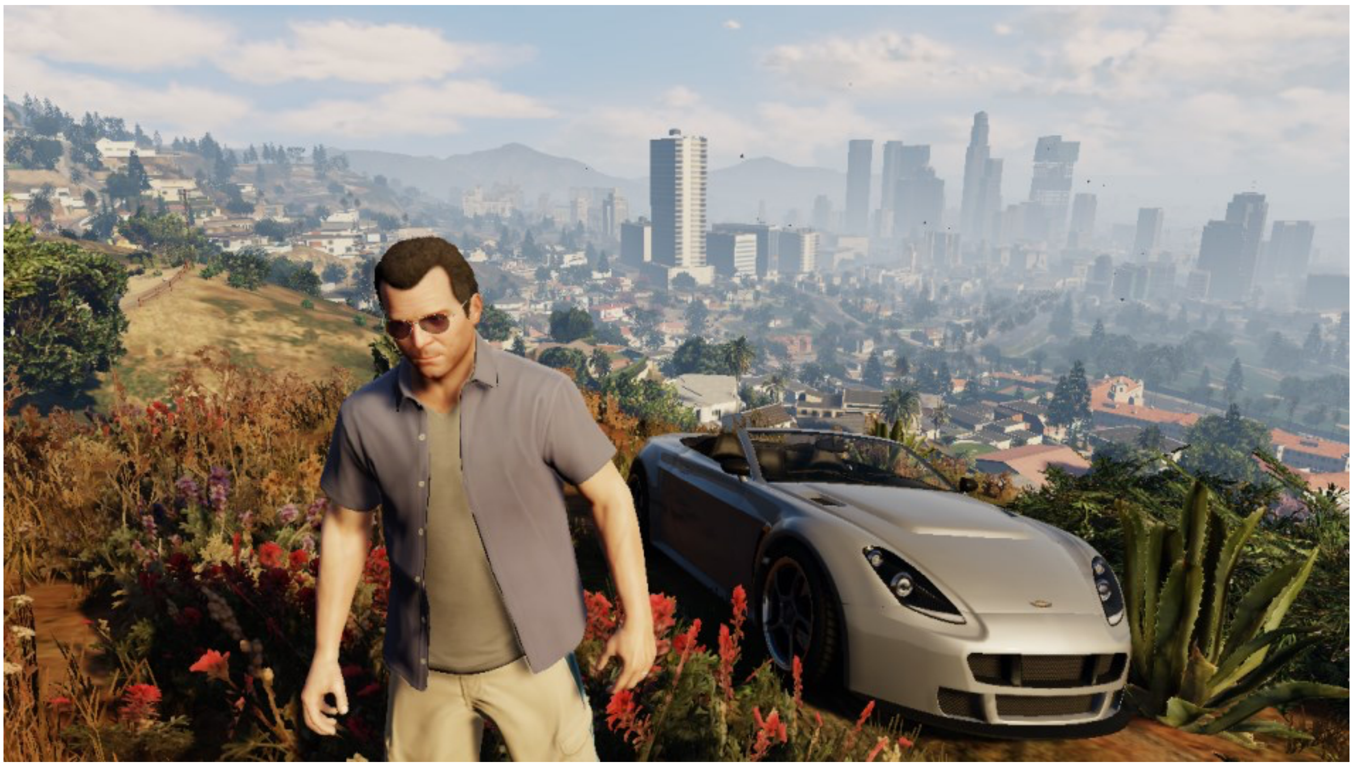
	
Cloud Density	**Cloud Normal**

Transparent Objects

This step renders all the transparent objects of the scene: the glasses, the windshield, the dust particles flying around…



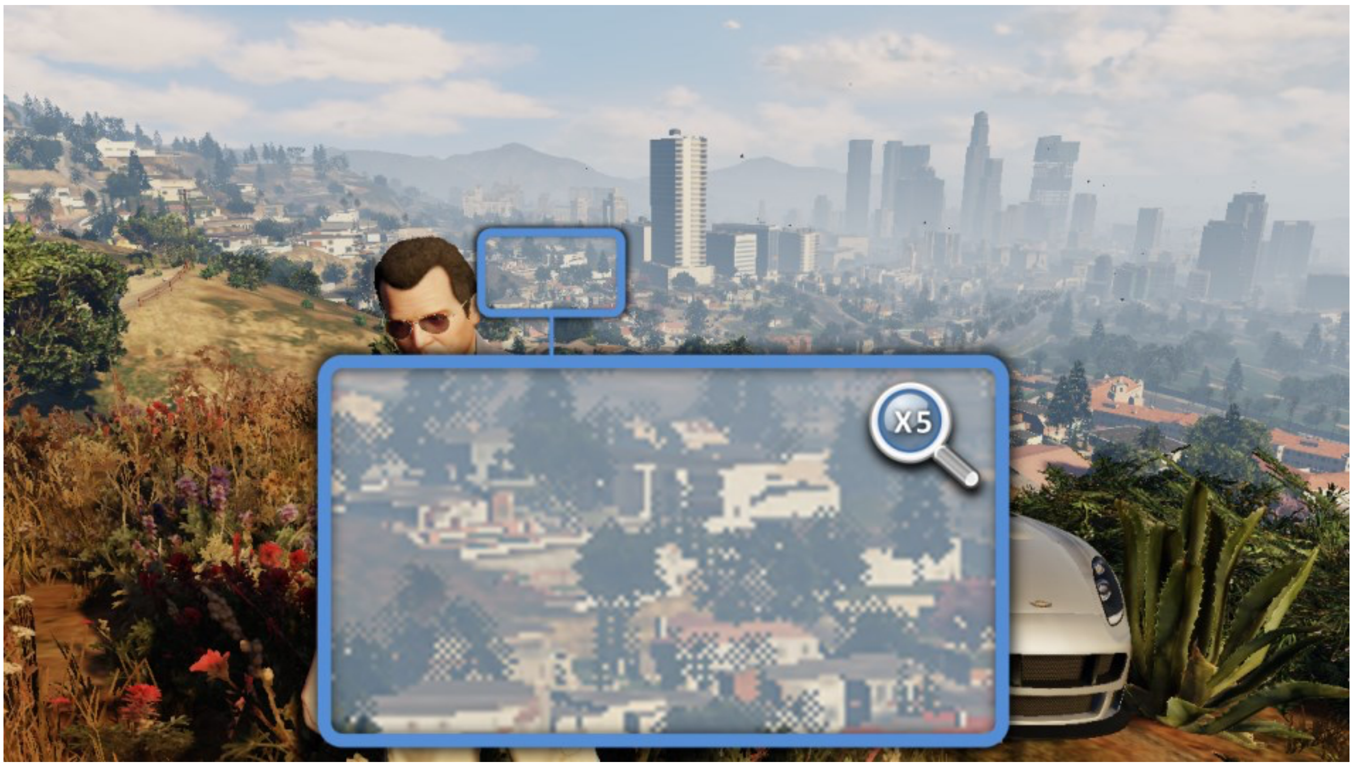
Transparent Objects (Before)

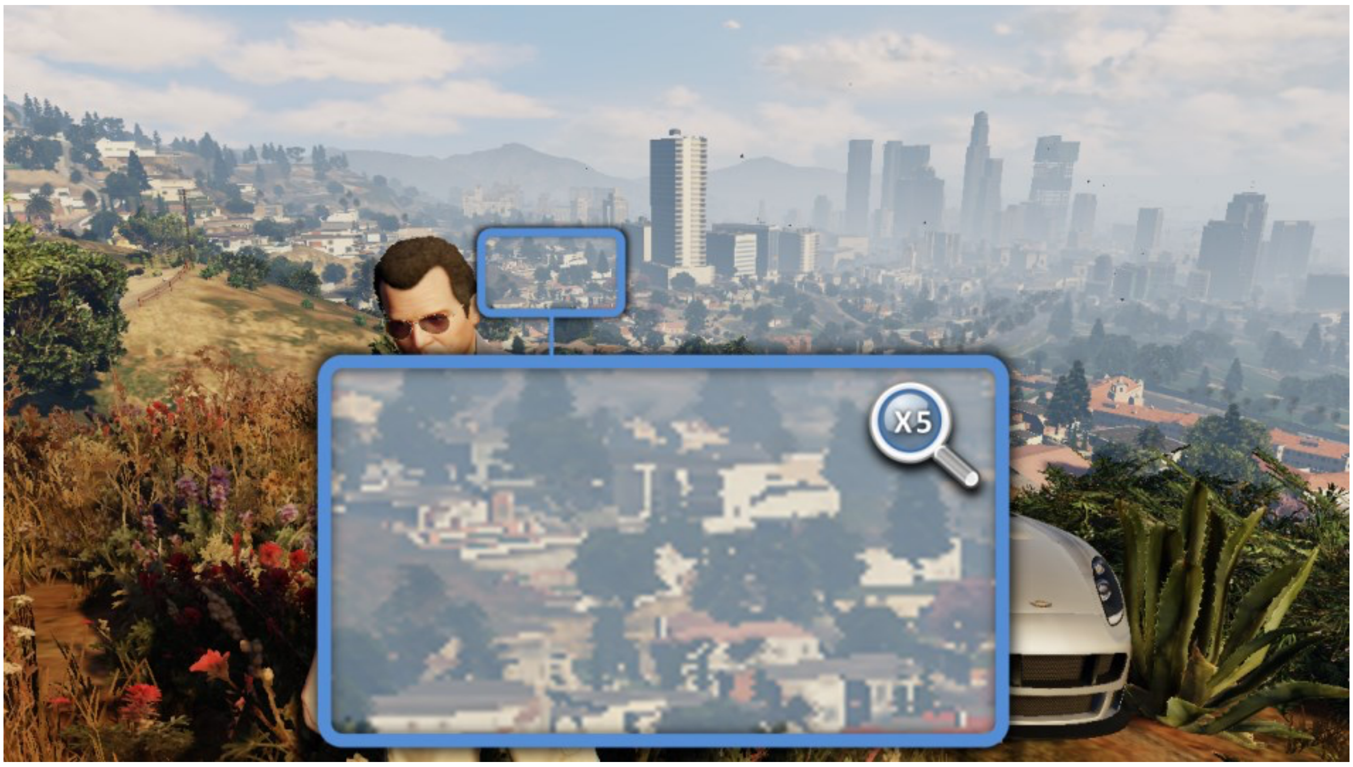

Transparent Objects (After)

All this is performed in only 11 draw-calls, instancing being heavily used for particles.

Dithering Smoothing

Remember our small digression previously about some trees being dithered in the diffuse map?
It’s time to fix these artifacts: a post-process effect is performed with a pixel shader, reading the original color buffer and the alpha channel of the diffuse map, to know which pixels have been dithered. For each pixel, up to 2 neighbor pixels can be sampled to determine the final “smoothed” color value.





It’s a nice trick because with just a single pass the image is “healed”: the cost is constant, it’s independent from the amount of geometry in the scene.
Note though that this filter is not perfect: both on PS3 and PC I noticed some checkerboard pattern on the screen that the filter didn’t catch in some situations.

Tone Mapping and Bloom

Our render image up until now has been stored in HDR format: each of the RGB channels is stored as a 16-bit float. This allows to have huge variations in the light intensity. But monitors cannot display such a high range of value, they only output RGB colors with 8-bit per channel.
Tone Mapping consists in converting these color values from an HDR to a LDR space. There are several functions which exist to map a range to another. A classic one which is widely used is *Reinhard* and it's actually the one I used when generating all the previous screenshots, it gives results close to the final render of the game.

But does GTA V really use *Reinhard*? Time to reverse some shader bytecode again:

Fragment Shader Assembly - Tone Mapping

```
// Suppose r0 is the HDR color, r1.xyzw is (A, B, C, D) and r2.yz is (E, F)
mul r3.xy, r1.wwww, r2.yzyy             // (DE, DF)
mul r0.w, r1.y, r1.z                    //  BC
[...]
div r1.w, r2.y, r2.z                    // E/F
[...]
mad r2.xyz, r1.xxxx, r0.xyzx, r0.wwww  // Ax+BC
mad r2.xyz, r0.xyzx, r2.xyzx, r3.xxxx  // x(Ax+BC)+DE
mad r3.xzw, r1.xxxx, r0.xxyz, r1.yyyy  // Ax+B
mad r0.xyz, r0.xyzx, r3.xzwx, r3.yyyy  // x(Ax+B)+ DF
div r0.xyz, r2.xyzx, r0.xyzx            // (x(Ax+BC)+DE) / (x(Ax+B)+DF)
add r0.xyz, -r1.wwww, r0.xyzx           // (x(Ax+BC)+DE) / (x(Ax+B)+DF) - (E/F)
```

Well well well… what do we have here?
`(x(Ax+BC)+DE) / (x(Ax+B)+DF) - (E/F)` is the typical equation of a tonemapping operator John Hable used in 2009 at Naughty Dog. It is a filmic tonemapping technique developed by Haarm-Pieter Duiker in 2006 while at EA.
It turns out GTA V doesn't use *Reinhard* but the *Uncharted 2* operator which doesn't desaturate the black areas as much.

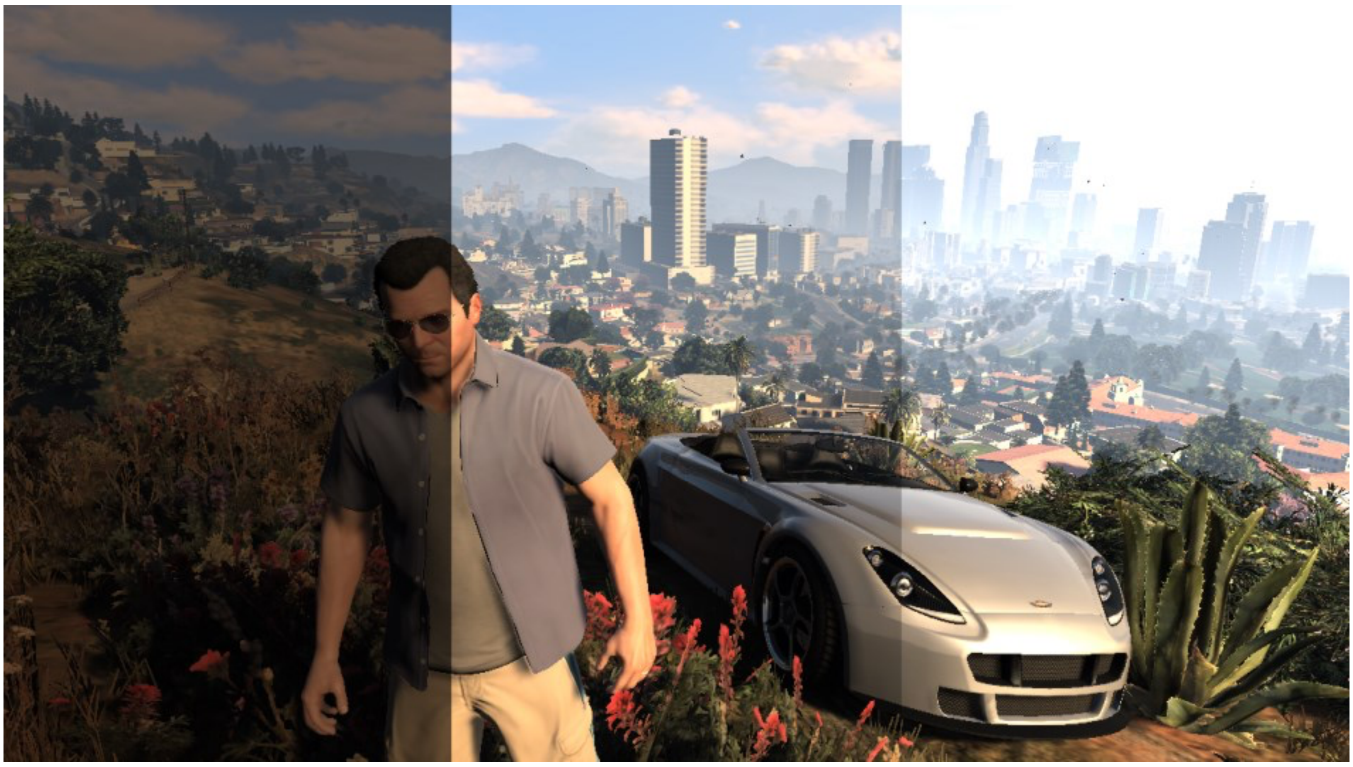
The process to convert to LDR is as follows:

- the HDR buffer is downscaled to ¼th of the resolution.
- a compute shader is used to calculate the average luminance of the buffer, outputting the result to a 1x1 texture.
- the new exposure is calculated, it controls how bright/dark the scene appears.

- a bright-pass filter is used to extract only the pixels having a luminance higher than a certain threshold (determined from the exposure).
  In this scene only a few pixels survive the filter: some spots of the car which have a strong reflected luminance.
- the brightness buffer is repeatedly down-sized and blurred up to 1/16th of the original size, then upscaled again several times until it reaches back half its original size.
- The bloom is added to the original HDR pixels and then the *Uncharted 2* tone-map operator converts the color to LDR, applying a gamma correction at the same time, leaving the linear space for the sRGB space.

The final result depends a lot on the exposure. Here are some examples illustrating the influence of this parameter:

Low Exposure	Medium Exposure	High Exposure

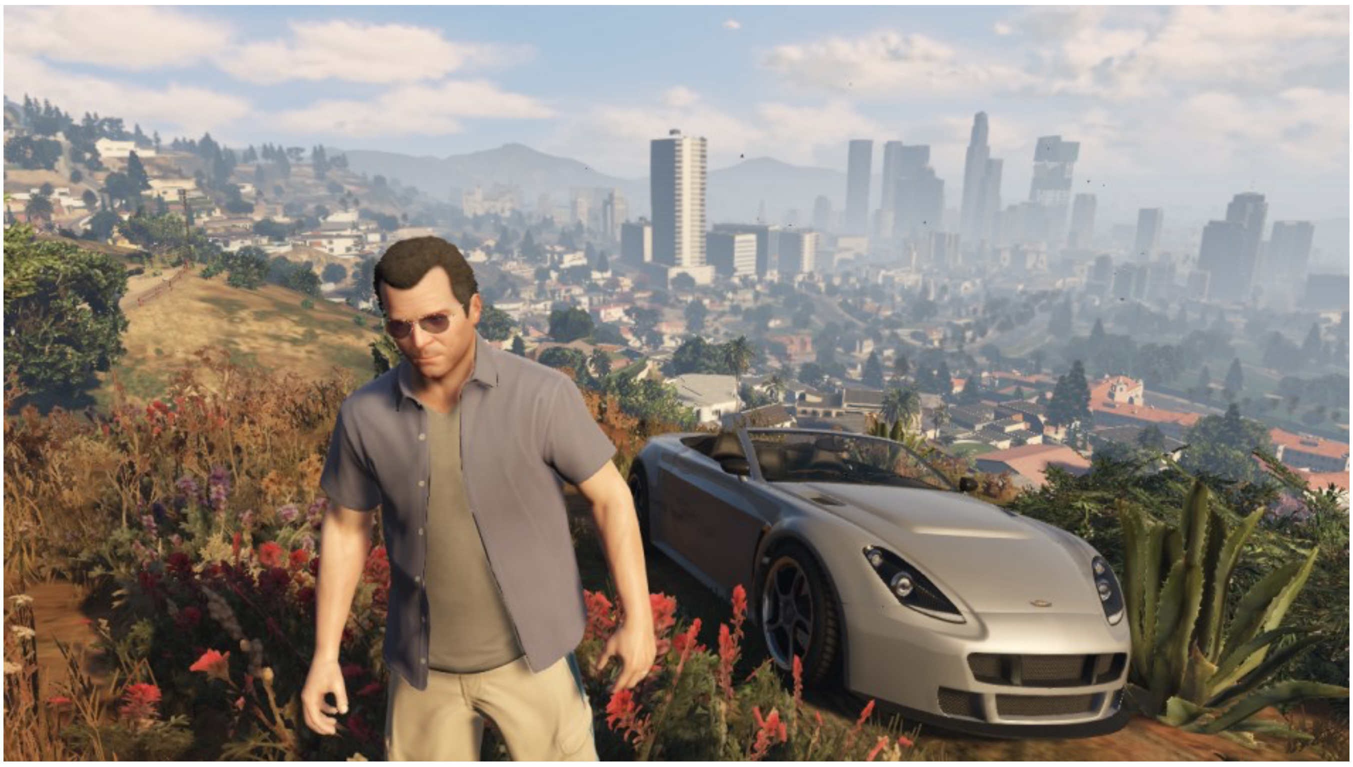


The exposure actually slowly evolves frame after frame, there are never abrupt changes.
This is meant to simulate the human eye behavior: have you noticed that after driving in a dark tunnel and suddenly exiting under the sunshine all

the environment looks too bright during a few frames? It then gradually goes back from “blinding” to “normal” while the exposure is adjusting to a new value. It seems GTA V even goes as far as giving “Dark → Bright” transitions a faster exposure adaptation than “Bright → Dark” ones, just like a human eye would behave.

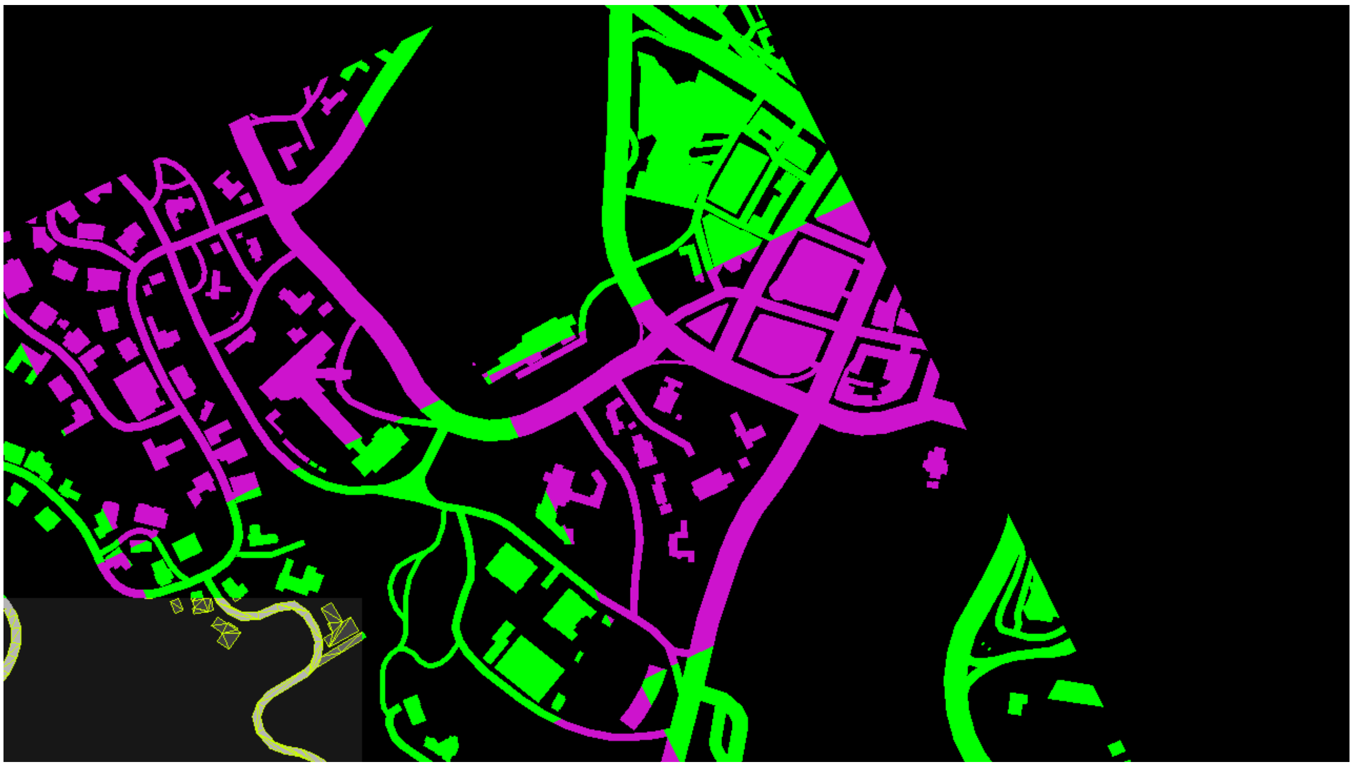
Anti-Aliasing and Lens Distortion

If the anti-aliased method used is FXAA, it is now performed to smooth-out the jagged edges of the meshes.
Then, to simulate a real-world camera, a lens-distortion is performed on the image by using a small pixel shader. It does not only distorts the image, it also introduces small chromatic aberrations on the edges of the frame, by slightly distorting more the red channel compared to the green and blue ones.



UI

Last touch: the UI, which here simply consists in the mini-map at the bottom-left corner of the screen. The map is actually divided into several square tiles, the engine draws only the set of tiles which potentially appear on the screen. Each tile is drawn through one draw-call. I colorized the tiles so you can see the structure better:

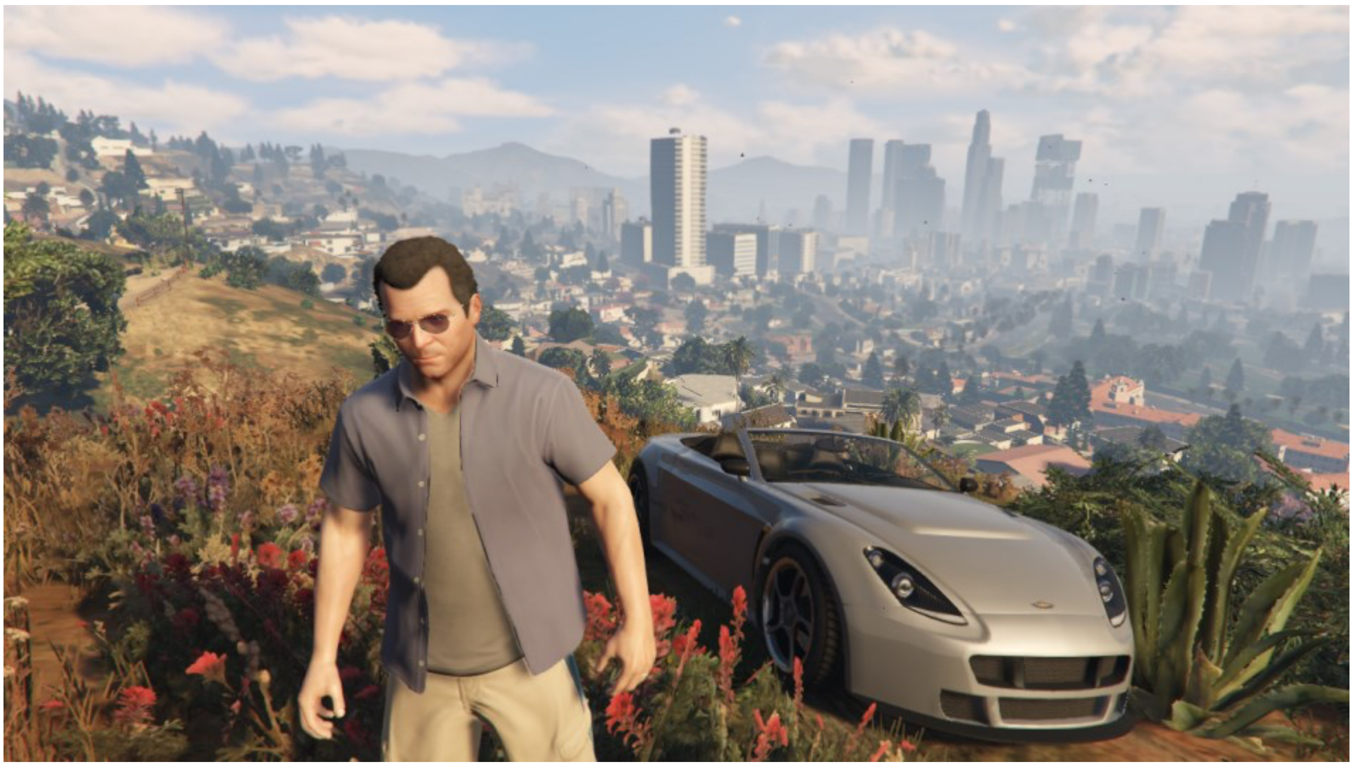


Mini-Map

The scissor test is enabled to only allow the rendering in the bottom-left corner and discard everything outside of it. All the roads are actually vectorized (see the wireframe in the screenshot above), they are rendered as meshes and can look great at pretty much any level of zoom.

The mini-map is then drawn into the main image buffer, and a few small icons and widgets are added on the top of it.

Well, it took a while but here you have it: the final frame in all its glory! All in all there were 4155 draw calls, 1113 textures involved and 88 render targets.



GTA V - Graphics Study - Part 2

Posted by Adrian Courrèges   Nov 2nd, 2015

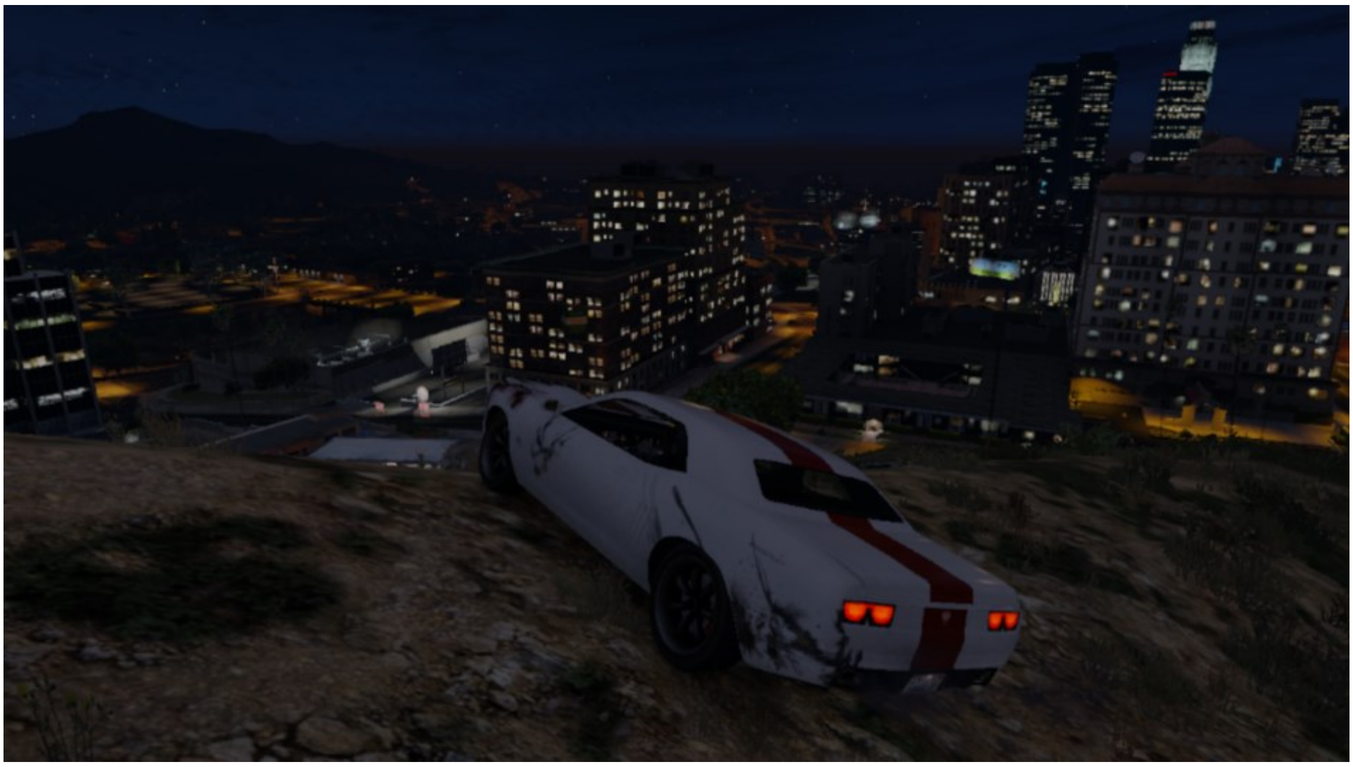
Level of Detail

If there's one domain where Rockstar outperforms the competition, it's when it comes to LOD. The world of *Los Santos* exists in so many different versions with more or less details/polygons, everything being streamed live while you play without a blocking loading screen. And it makes the whole experience so much more immersive.

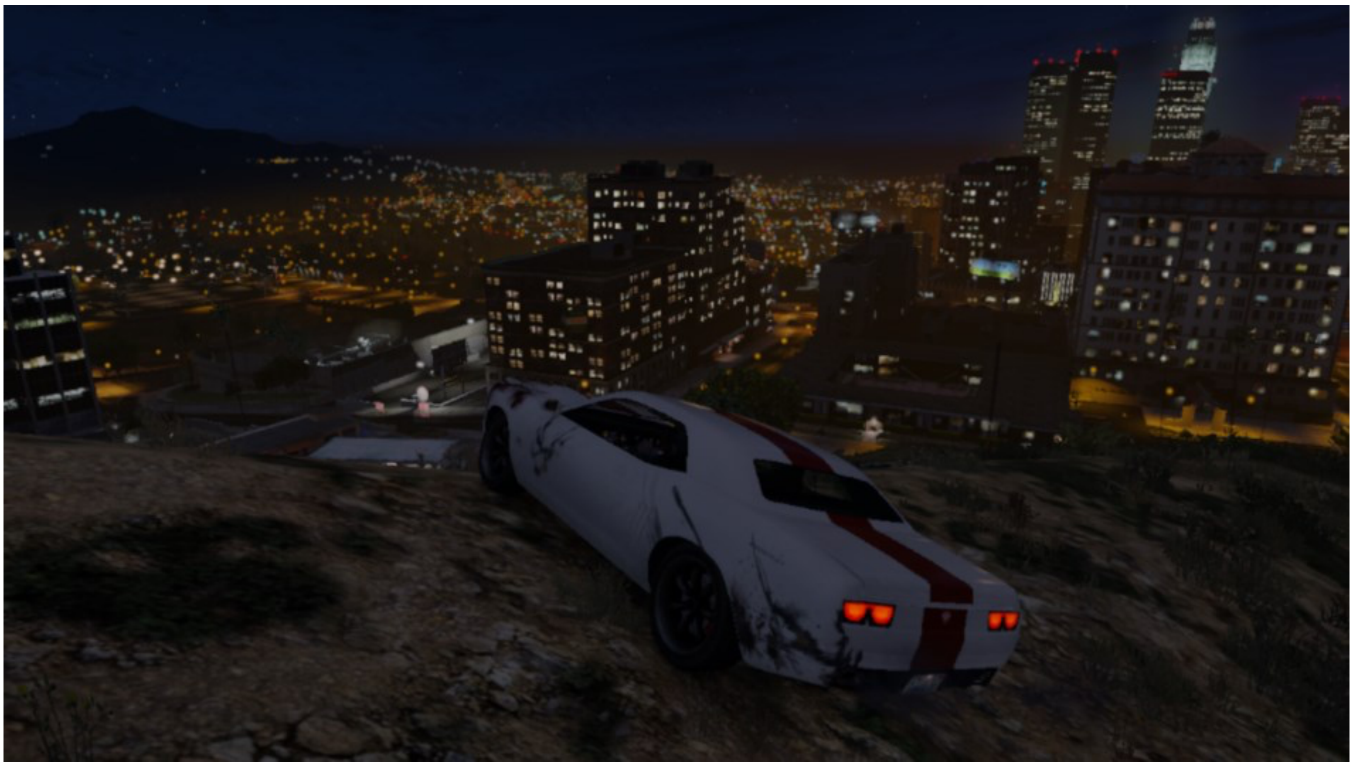
Lights

> *All the little lights you see in the far distance are real, you can drive towards them and find the bulb that casts the light.*
>
> ***Aaron Garbut***

This is what Aaron Garbut, one of the Rockstar North founders and art director, was declaring shortly before the PS3 release.
How accurate is his statement? Let's consider this night scene:



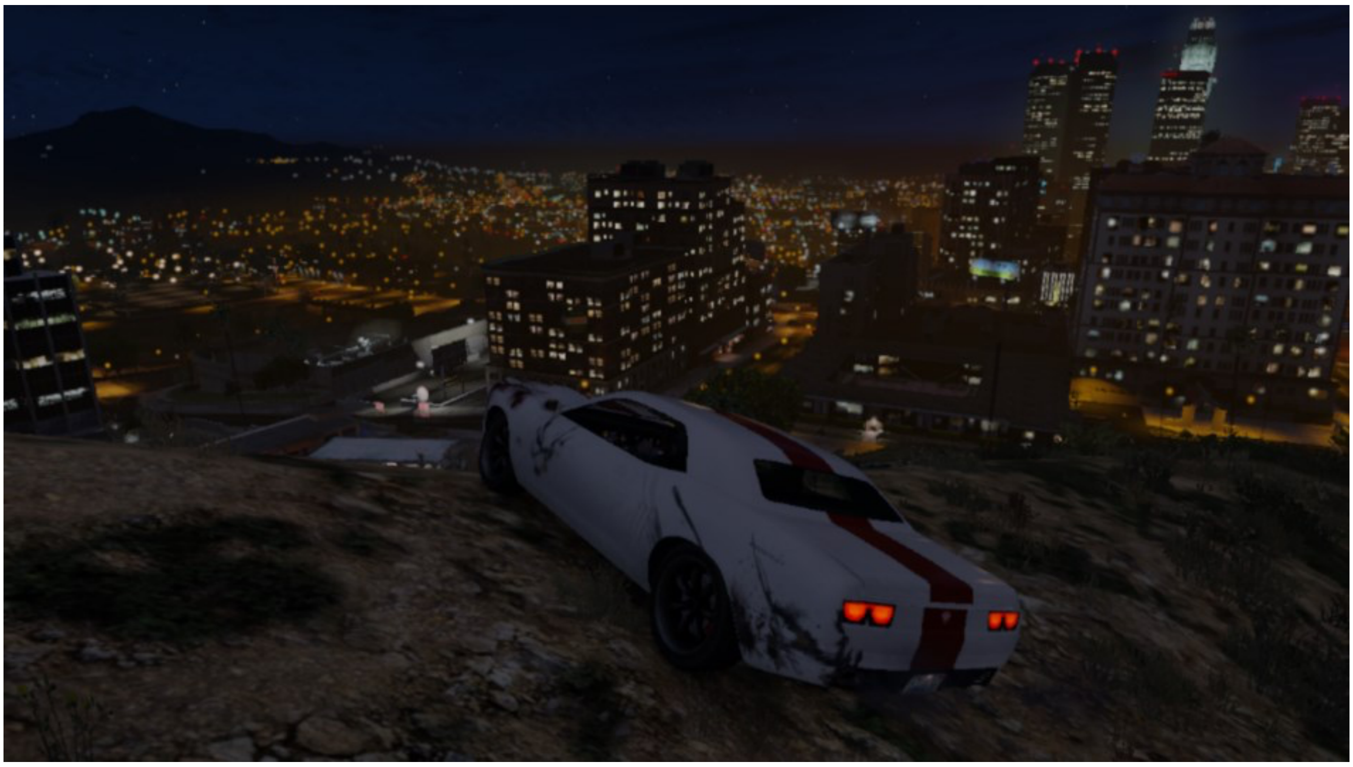
Light Bulbs (Before)



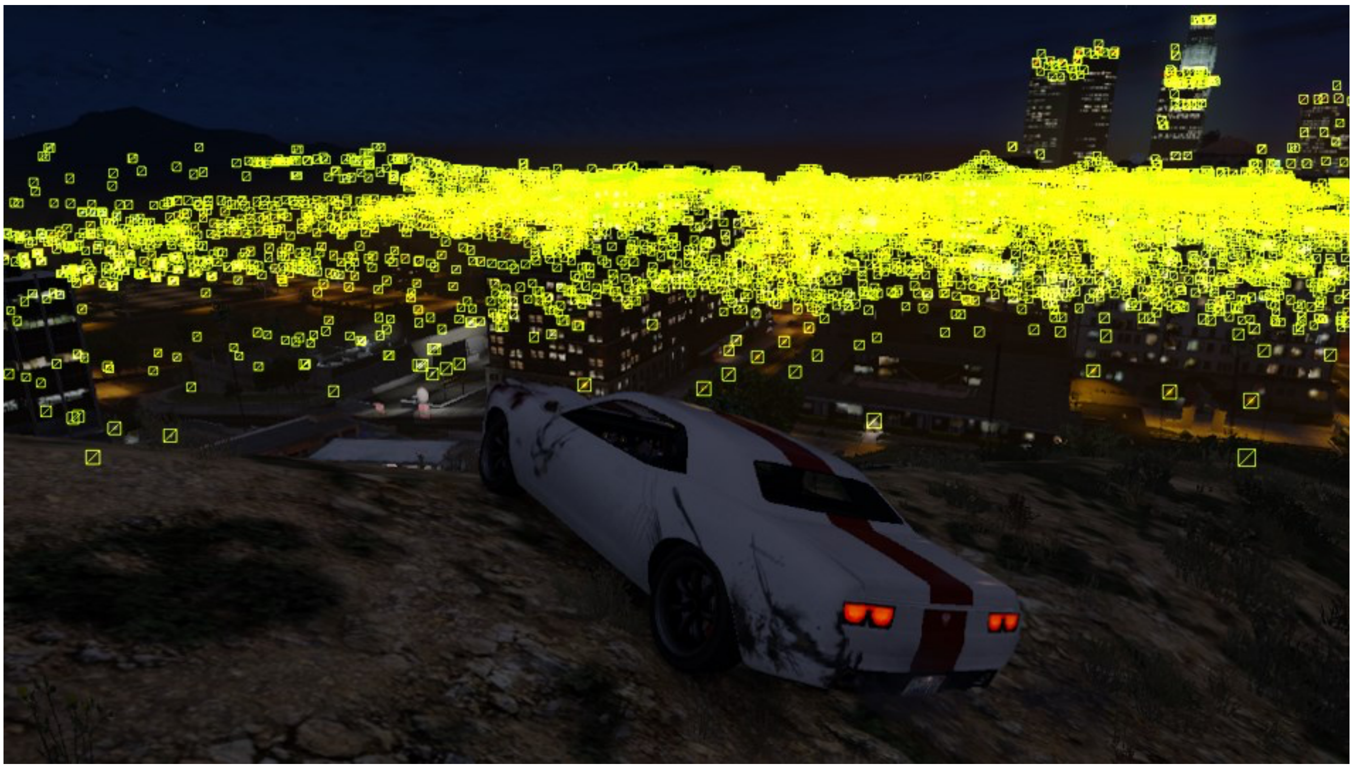
Light Bulbs (After)

Well it simply is the truth: every single small light spot you can see is a quad rendered with a small 32x32 texture like the one on the right.

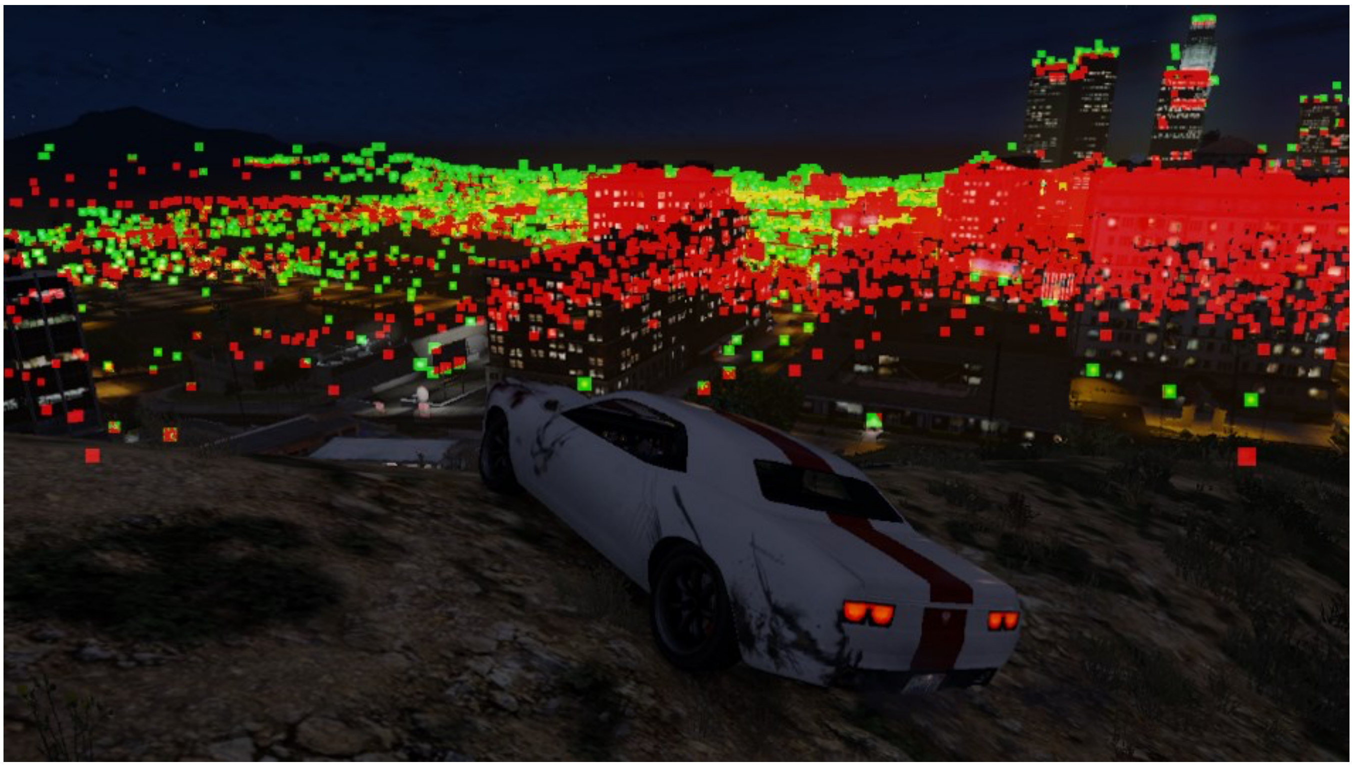
They are all heavily batched into instanced geometry but still it represents tens of thousands of polygons pushed to the GPU.



Light Bulbs



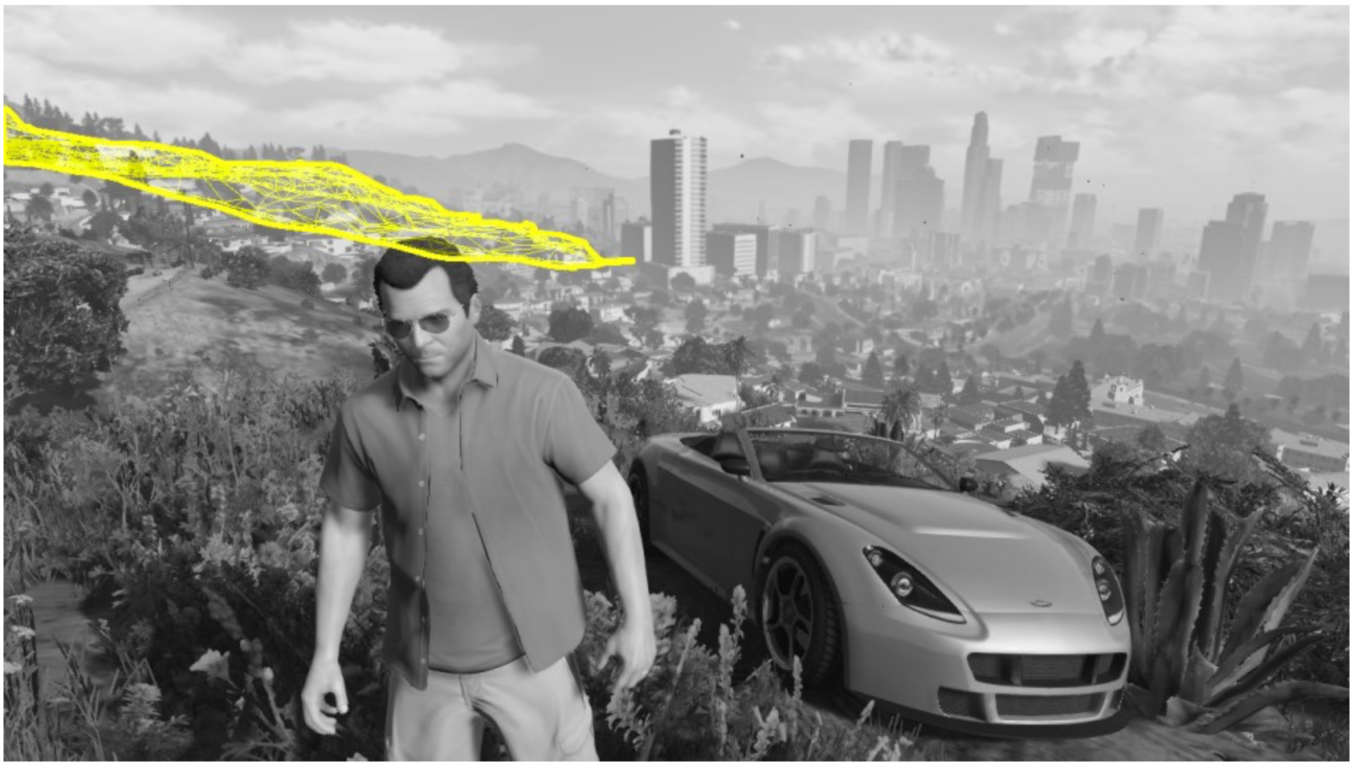
Light Bulbs - Wireframe


Light Bulbs - Depth Test Pass/Fail

And these are not just static geometry: the car headlights are also moving along the roads, updated in real-time. Of course at this distance, no need to render the full car models, only 2 headlights are enough to create the illusion. But if you decide to go near some distant light, as you come closer, the LOD increases and eventually the full model of the car is drawn.

Low-Poly Meshes

Let’s go back to the frame we dissected previously. Some really vast portions of the world are rendered in a single draw call. For example if we consider the rendering of the hill below:



Hill rendered in a single draw-call

So what exactly is that tiny hill far away?

Galileo Observatory

Turns out it's not small at all, it's actually ***Vinewood Hills*** a big area spanning across several square kilometers, with dozens of houses and buildings.
There's the *Galileo Observatory* sitting on the top of the hill, the *Sisyphus Theater*, *Lake Vinewood*, all of these areas which, when you explore them or drive by them, are rendered with thousands of draw-calls and tons of polygons.

But in our case, this zone is far away, so a low-poly version is rendered: a **single draw-call pushing only 2500 triangles**.

Low-Poly Model of *Vinewood Hills*

All the rendering is done with a single mesh reading from a diffuse texture. Even if some tools exist to convert a mesh to a lower-poly version, they can’t fully automate the process and I wouldn’t be surprised if the 3D artists at Rockstar spent days fine-tuning the meshes manually.

Another example, this time the *Little Seoul* district accounting for several blocks of the city, rendered with a single draw-call.

LITTLE
SEOUL

Low-Poly Model of *Little Seoul*

Having a low-poly version of the world is extremely useful to render efficiently a reflection cubemap which doesn't need to be neither very detailed nor high-resolution. For games with a single LOD level rendering such environment cubemaps realtime is very costly or simply impossible because of the high amount of geometry involved.

Asset Streaming

Creating several versions of a world like GTA at different LODs is a huge and time-consuming task, it's quite a challenge. But even if you reach this goal, you're still only half-way there: you may have gigabytes of models and textures on your hard-drive but they're all useless if you can't figure out a way to load them efficiently in RAM or on the GPU memory.

GTA V streams assets in real-time, loading/unloading models and textures as you move to another area of the map. The really impressive thing is achieving this in real-time while remaining stable for hours.

> *Technologically the biggest achievement has been squeezing this all into the console's memory and making it run as smoothly as it does. […] We can stream far more and compress far more into memory,*

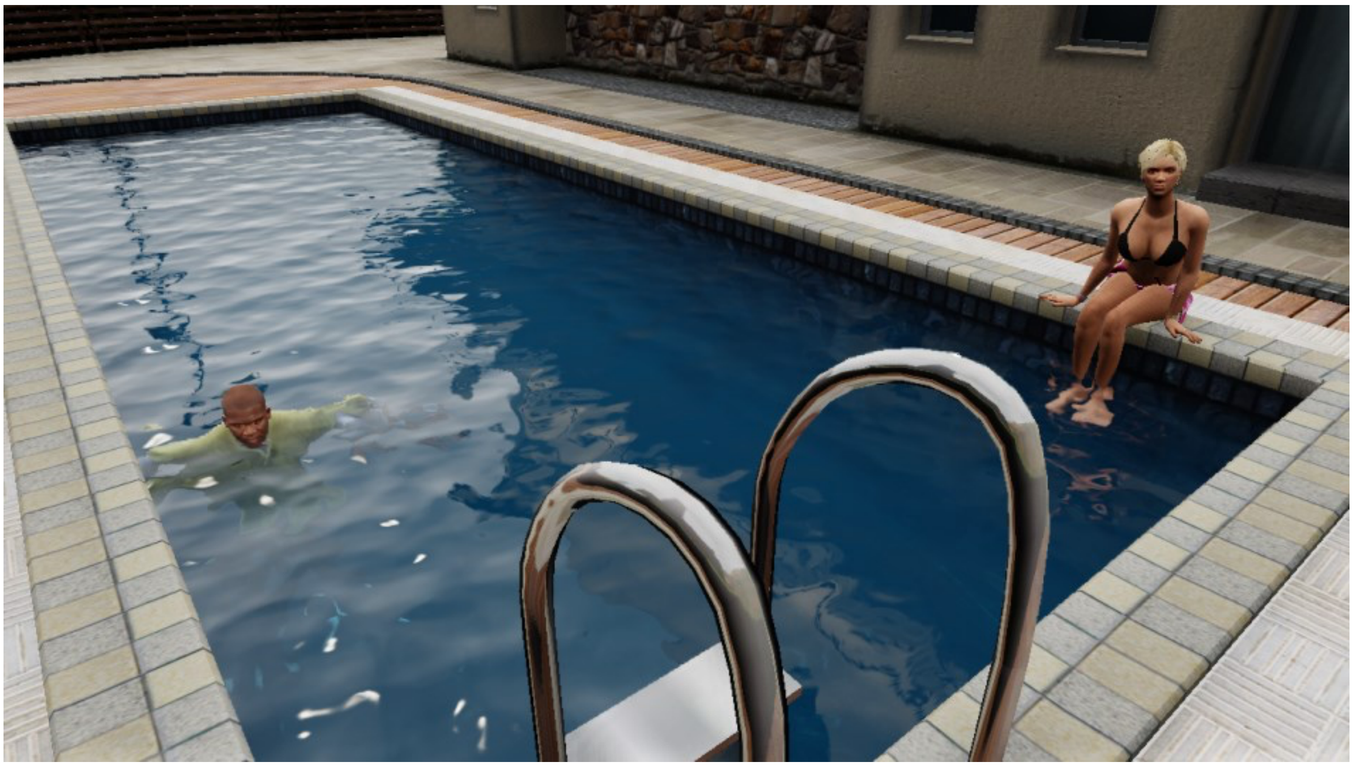
> *meaning orders of magnitude more detail than we had in GTA 4.*
>
> ***Aaron Garbut***

Of course this streaming system has its limitations: for example when you switch character, the camera jumps from one side of the map to another, in this case the streaming system is suddenly overloaded and it is perfectly understandable it needs 5 seconds to sort things out and catch-up. But GTA V handles this transition nicely with a zoom-out/translate/zoom-in animation, it doesn't feel like a loading screen at all.

When you drive a car normally, the speed you're moving at is slow enough that the streaming system has enough bandwidth to keep-up with the updates. This is not the case for planes: they are way too fast for the streaming system, that's why their speed has been greatly reduced on purpose compared to the real-world. When flying, meshes are also drawn at a lower LOD than when walking/driving to relieve the pressure on the streaming, but still there can be some "pop-in" of assets from time to time. On PC a special option has been added in the settings: *"High Detail Streaming While Flying"*.

Reflections

Since the frame dissected previously didn't have so much water, let's see more closely how a swimming pool or the ocean is rendered with the scene below:



Swimming Pool

Like we saw previously, the scene is rendered normally: an environment cubemap (see on the right) is generated, it is mainly used when rendering objects which reflect their surroundings.
For example the reflection of the pool ladder is achieved thanks to this cubemap.

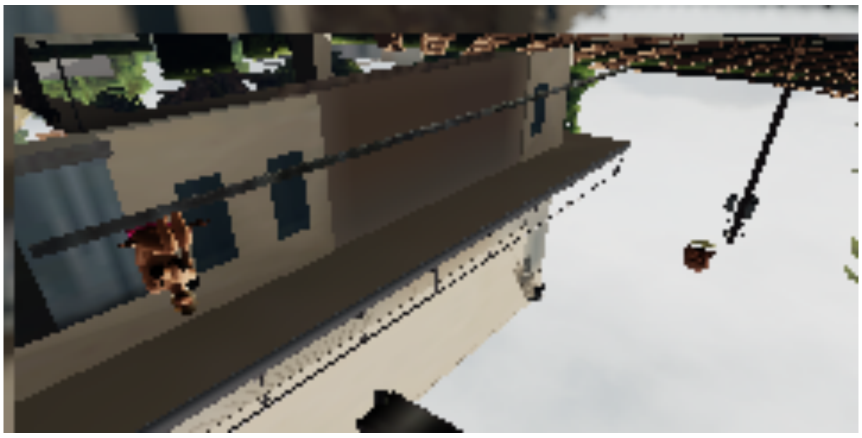
Note the absence of characters or small objects like the ladder itself in the cubemap.

This is how the scene looks like just before rendering the water:

Before rendering the water

Rendering the water surface is a whole different story, it doesn't involve the cubemap.

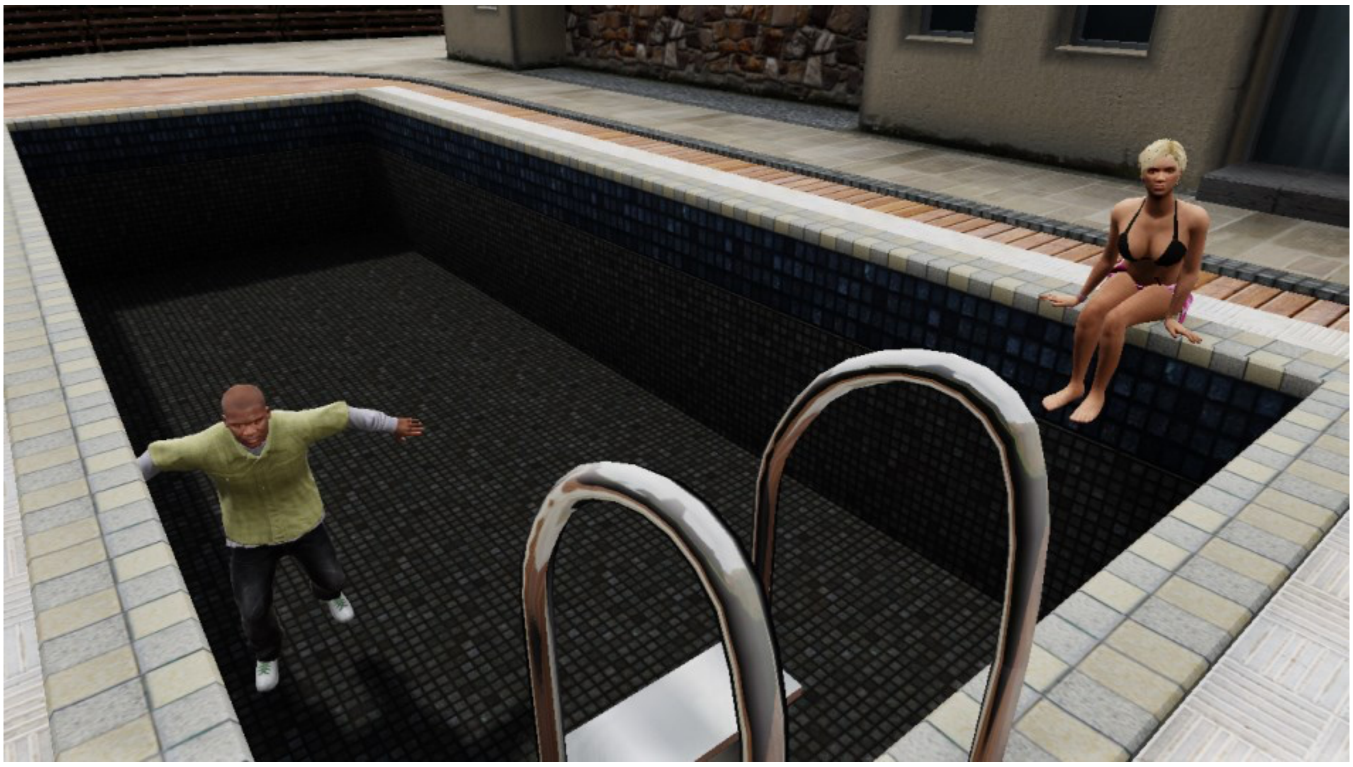
Reflection Map

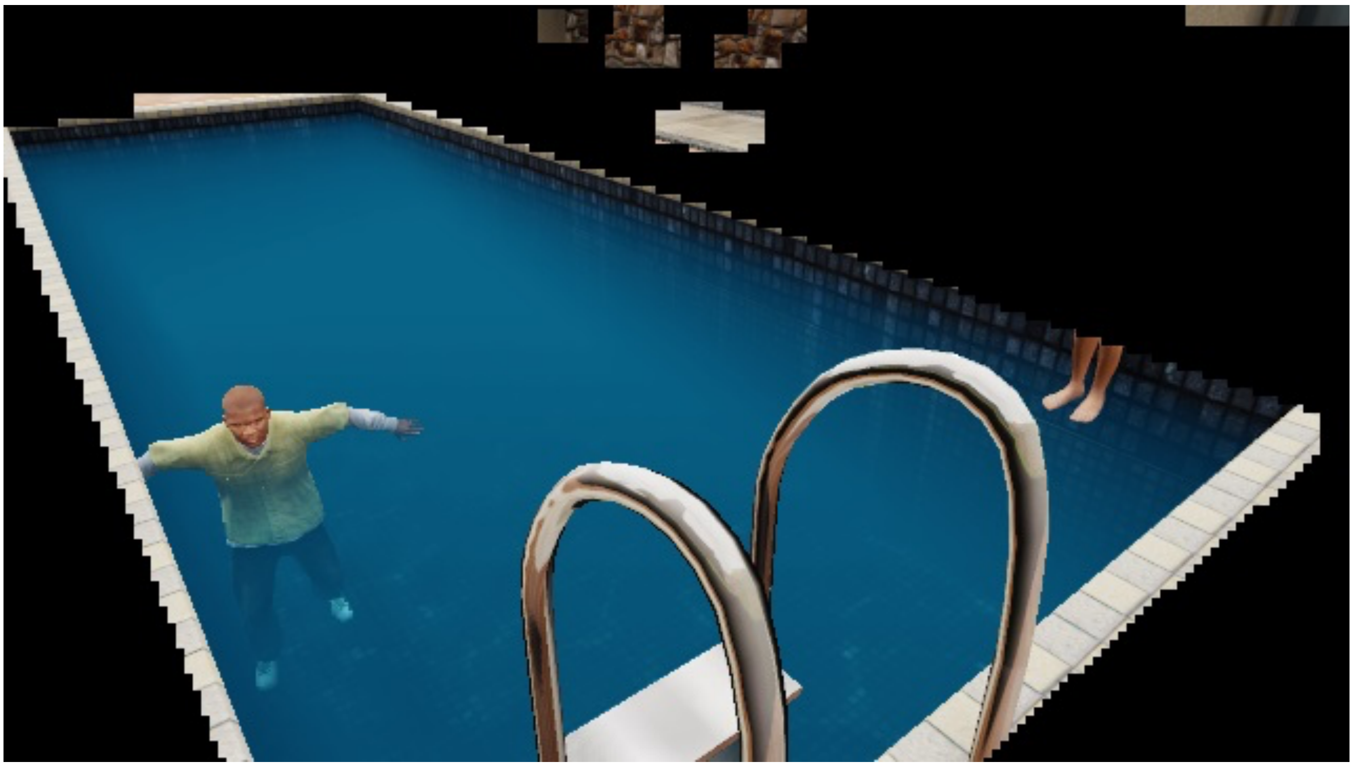


First, a "planar reflection" map is generated. It is very low-resolution, 240x120, and the process is similar to the one of the cubemap generation but this time just one buffer is generated and the ladder and the characters are present.

The scene is rendered upside-down, later when it is sampled a symmetry is also applied to get the correct reflection.

Refraction Map





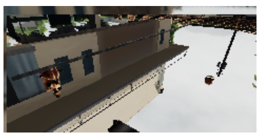
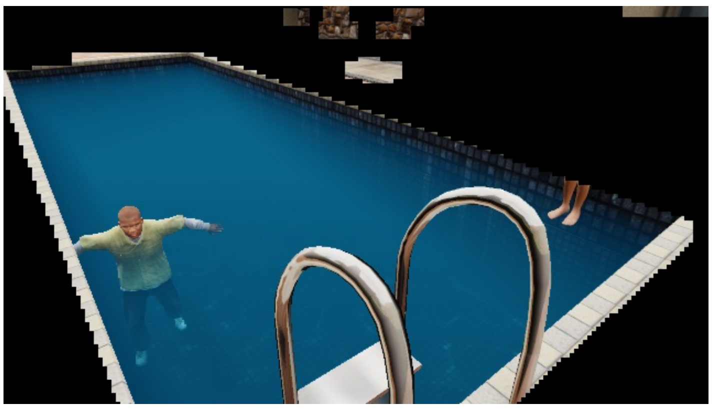
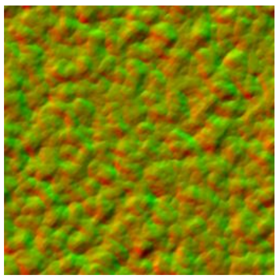
A subset of the image is extracted: the portions where the water surface is located in order to create a refraction map. Its goal is to simulate the refraction effect later: the light coming out from under the water surface.

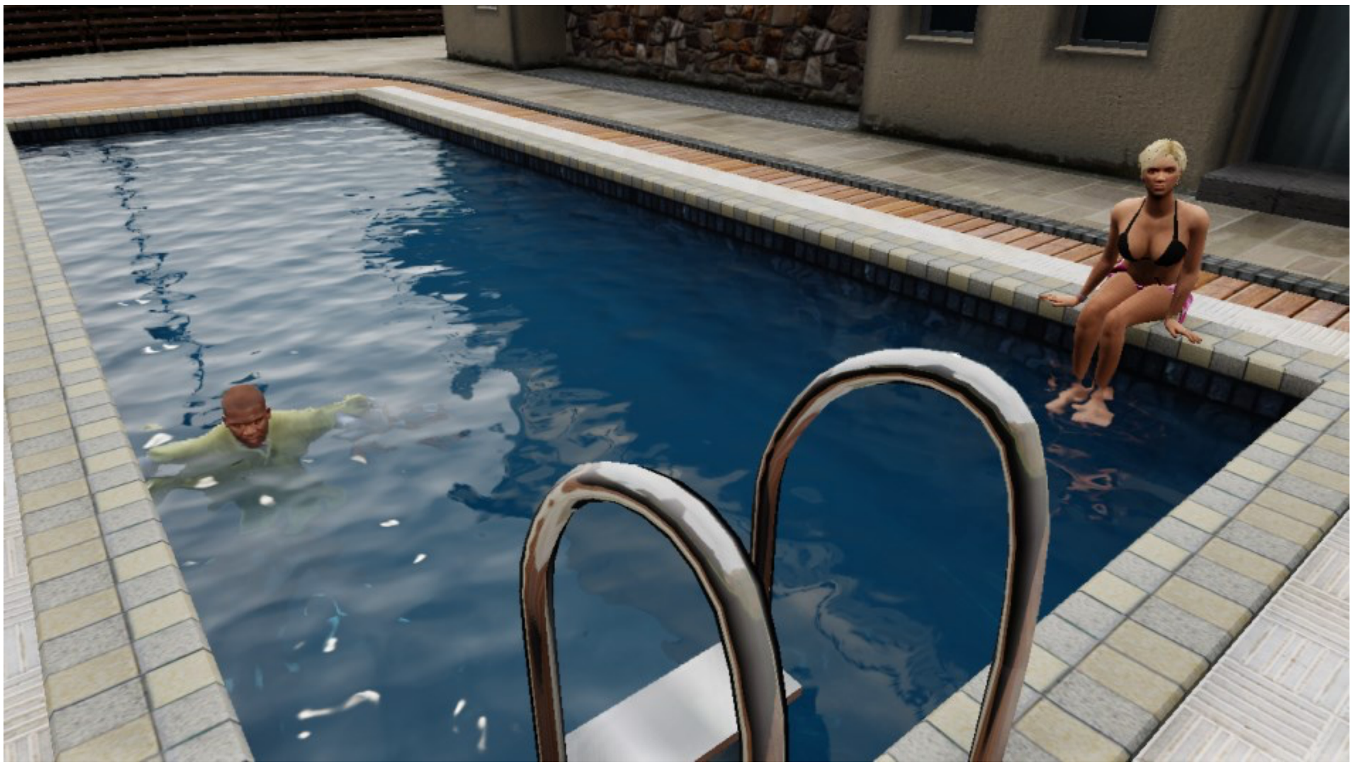
This is when some “water opacity” blue is added (the deeper the stronger the color), as well as caustics. The final map is half the size of the original buffer.

Combination

To combine the different buffers, a rectangular polygon is drawn to represent the pool water surface. The normals of the polygon are perturbed on a per-pixel basis to create the illusion the water is moving a bit, relying on some bump texture.
In the case of the ocean, it's not just the normals which are perturbed, instead of a quad, a whole mesh is drawn with the vertices updated at each frame to simulate moving waves.

Based on the normal of each pixel the pixel shader will fetch the reflection and refraction maps at different points, the tap coordinates being calculated through the Fresnel equations.

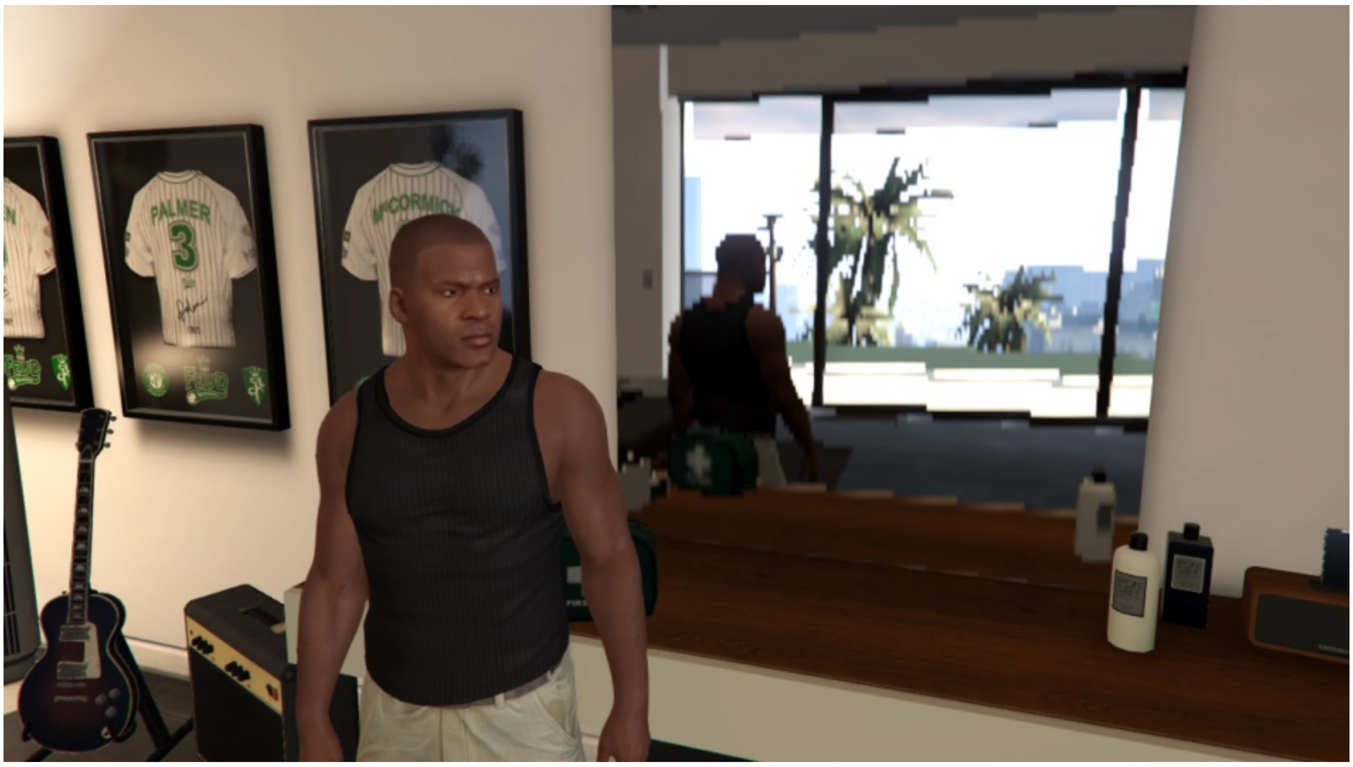
 **Reflection Map**	 **Refraction Map**	 **Bump Map**



The final result is quite nice. Coupled with a realistic perturbation of the water surface the effect is very convincing.

Mirrors

Mirrors are rendered with exactly the same technique as the water, it's even easier actually since mirrors only reflects light, there is no refraction coming from behind the surface to take into account.

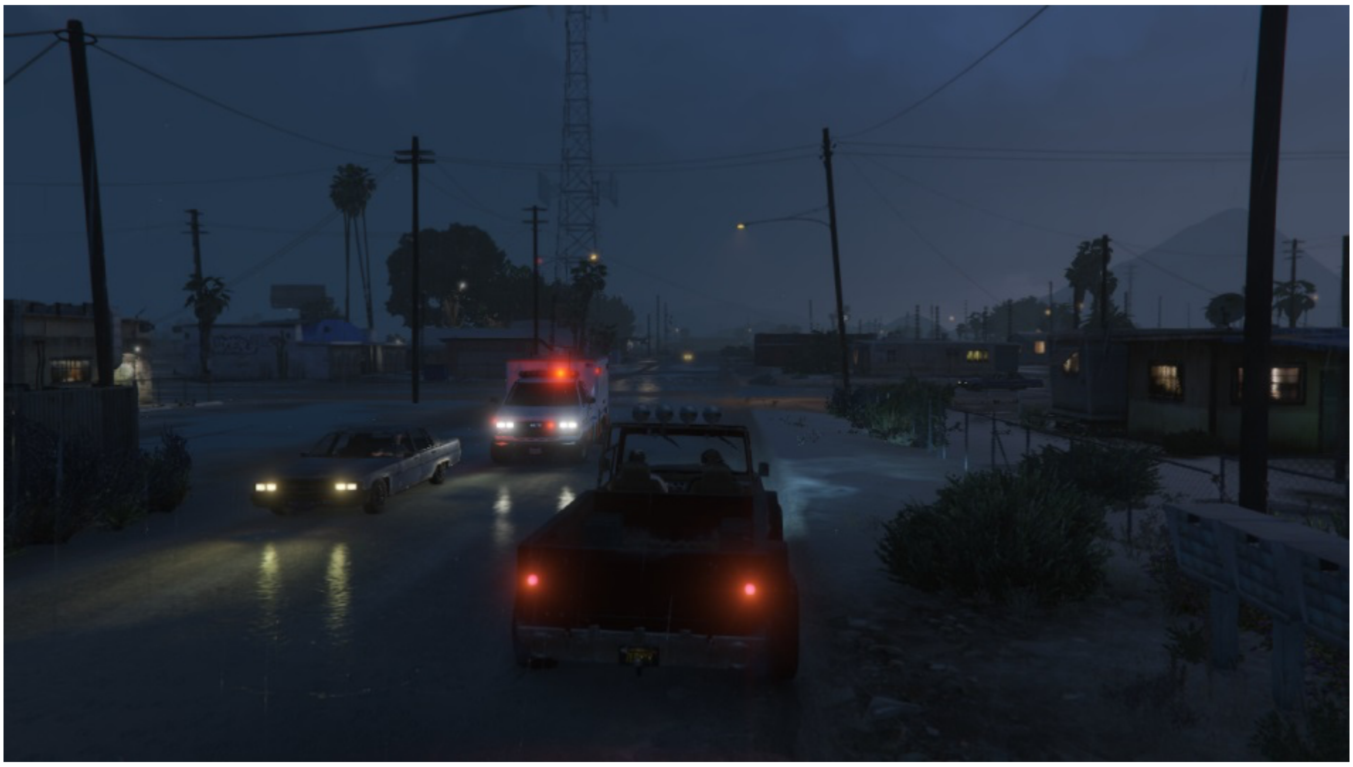


Mirror Reflection

Unlike water, the mirror surface is perfectly flat and still, making it harder to hide the fact it's pretty low-resolution – texels are quite visible. Increasing the reflection quality in the settings can give it a higher resolution.
Generating the reflection map requires an extra pass rendering the scene and it can be quite heavy. The engine avoids performing this extra pass if the mirror is not in the viewport, or when the player is too far away from it (in which case it just appears as a black quad).

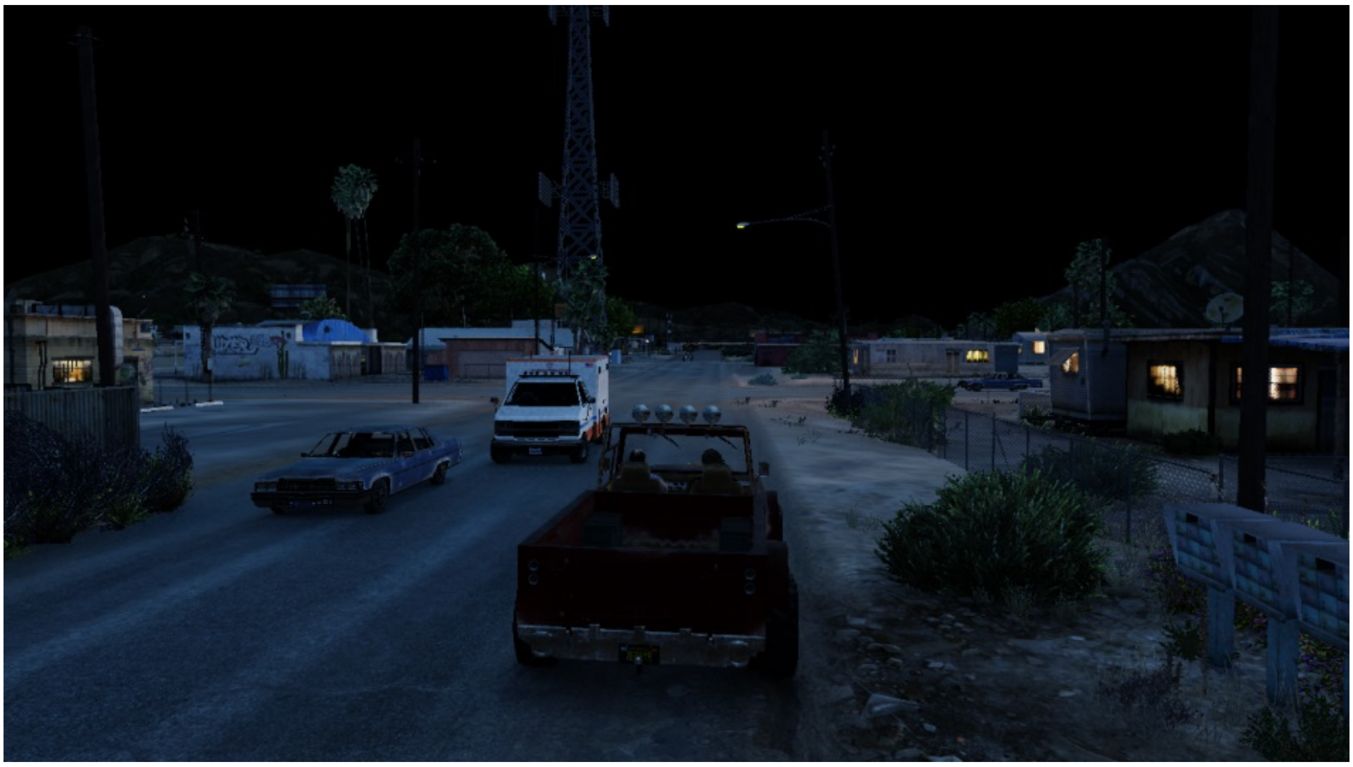
Spotlights

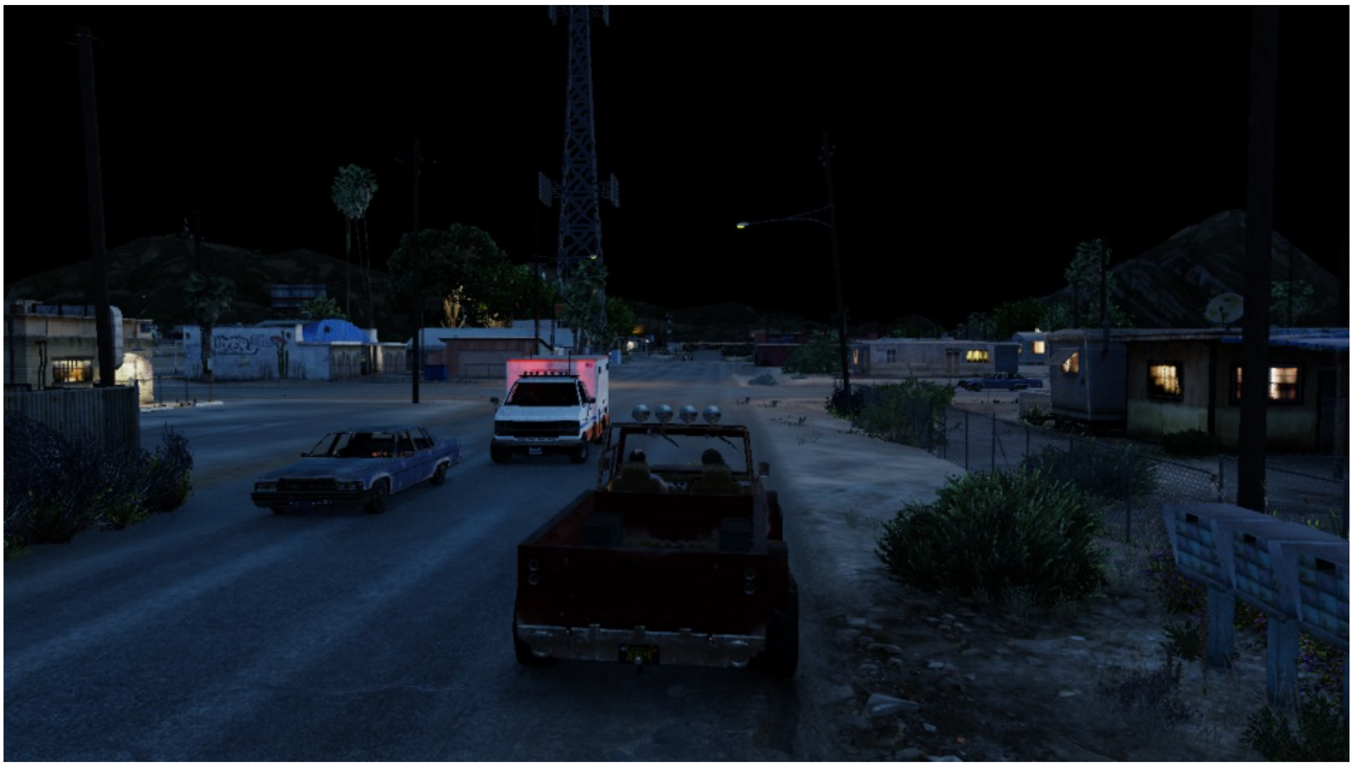
Remember the environment map generated at the beginning of each frame doesn't contain any character or car, just the main buildings and scenery? In this case how can the car headlights be reflected on the wet road in the screenshot below?

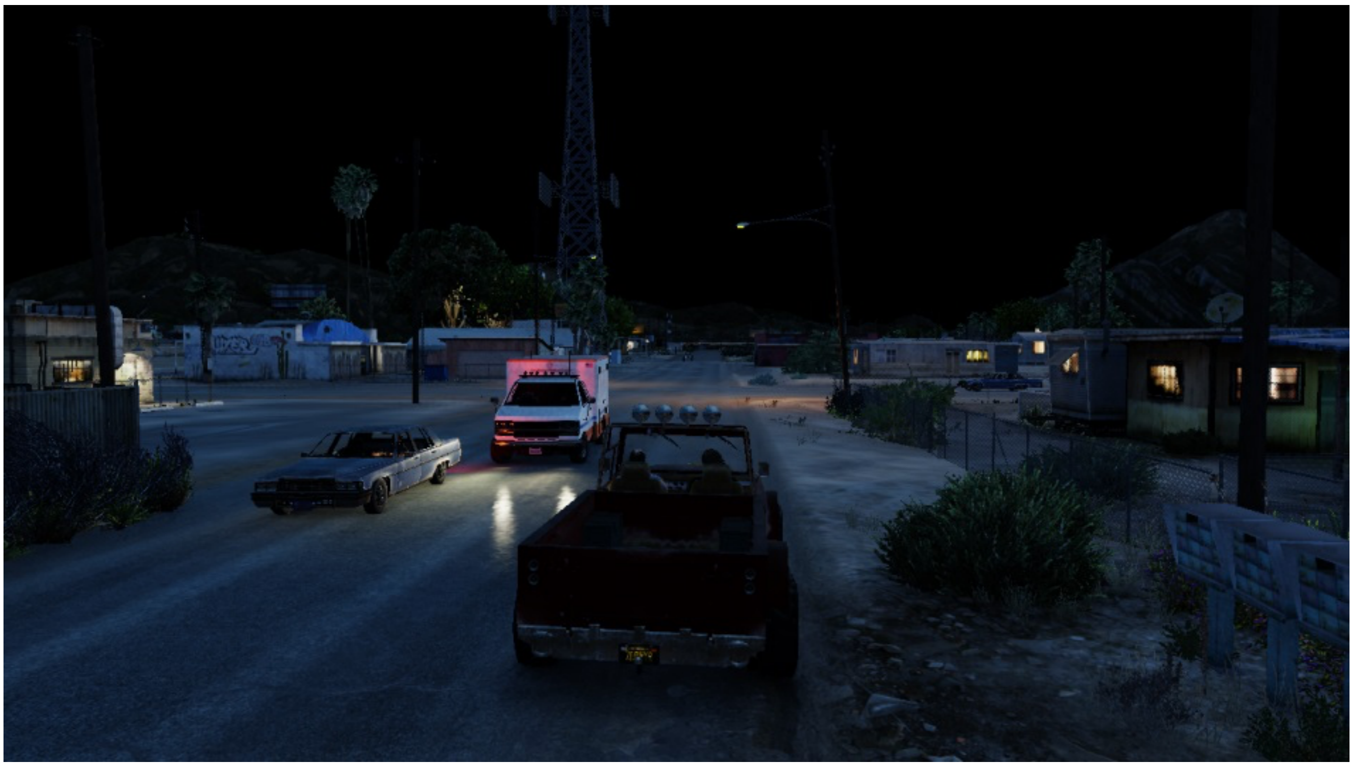


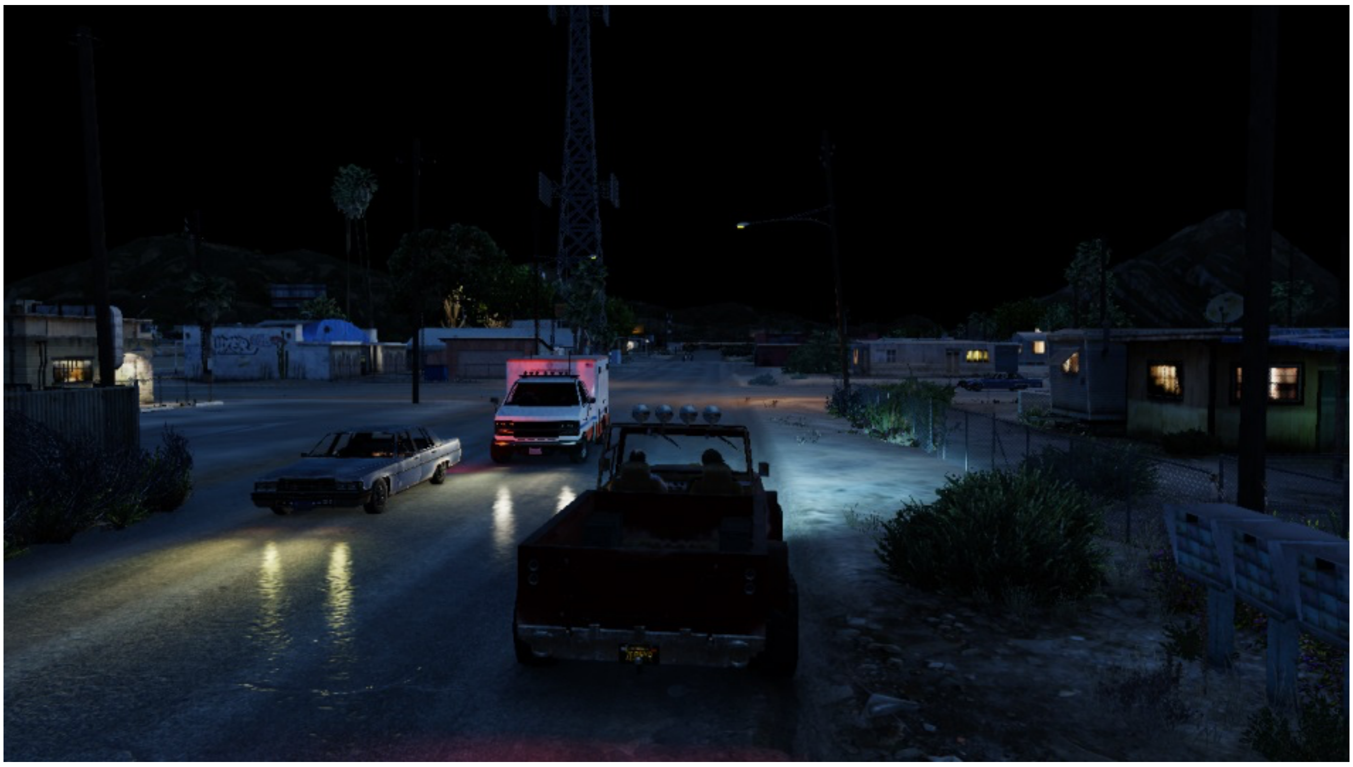
Headlight Reflections

It was not really obvious in the frame we dissected in *Part 1*, because it was a day-time scene but actually right after all the G-Buffers have been combined, the lights are drawn one by one. For each bulb, the light cast on the other meshes is calculated, including the strong reflection highlights on surfaces with high glossiness like the wet road.

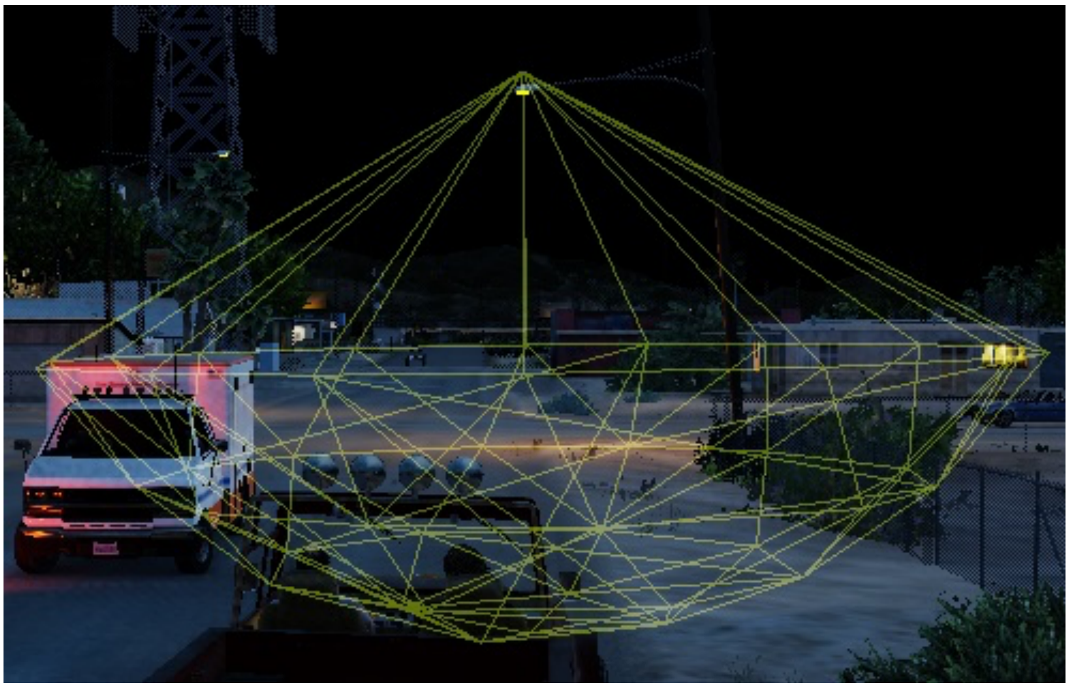








For each light source, a mesh is drawn: it's originally similar to a tessellated octahedron but it's modified by a vertex shader to match the shape of the light halo.



The mesh is not textured, the goal is just to touch all the pixels inside the light halo so a pixel shader can be invoked. The shader will dynamically calculate the lighting depending on the pixel depth, its distance from the light source, its normal, its specular/glossiness properties.

On the right is a wireframe view of the mesh used to calculate the influence of the lamppost light on the road.

Here you can see one big advantage a deferred pipeline has over a forward one: a large number of lights can be rendered in the scene, with pixel shaders invoked as little as possible, involving only the pixels really affected by the light source. In a forward pipeline, you would have to calculate at once the influence of many lights for every fragment even if the fragment in question is not actually affected by any light or end up later hidden by another one.

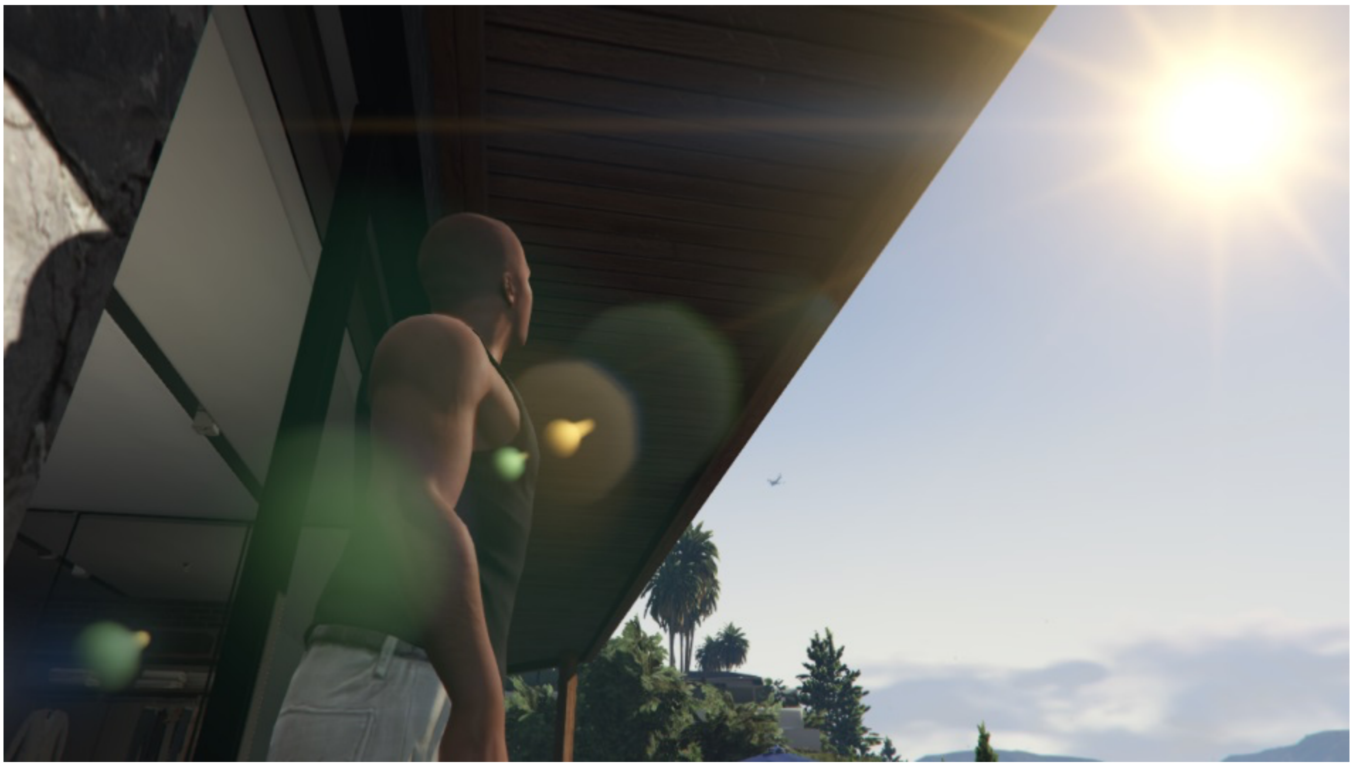
- **Part 1: Dissecting a Frame**

Post Processing Effects

Post-effects are performed once the scene has been rendered, to enhance, fix artifacts, change the mood…
We saw in *Part 1* how a few post-effects were applied, like bloom, anti-aliasing or tone-mapping. But there are several other effects used in GTA V.

Lens Flares & Light Streaks

When the light goes through a real-world lens, the scattering and internal reflections sometimes cause artifacts.
What I call "lens flares" here is a collection of bright spots along the axis defined by a bright light source and the center of the screen. There are also "light streaks" which are rays originating from the light source. These artifacts are very common in movies, when they are added to a game frame they can give a kind of "cinematic" feeling.



Lens Flares & Light Streaks

There are usually 2 ways to render such artifacts:

- *image-based*: extracting the brightest areas, duplicating and deforming them. Works for any number of bright light sources.
- *sprite-based*: adding textured-sprites and managing their positions manually. Each light source must be handled separately but artists can have more control over the artifact shape, color, intensity…

GTA V actually uses both techniques: the image-based approach is used to add a subtle blue halo at the bottom-left corner of the image, it's actually a symmetric of the bright-pass buffer. But the most visible artifacts in this scene result from the sprite-based approach, it is applied for the sun only. First, light streaks are added by rendering 12 rotated quads centered around the sun. Then for the lens flares, 70 sprites are drawn along the axis "sun – screen center". Artifacts come closer to each other as the camera points towards the sun.

There are several sprites the engine uses to simulate different lens artifacts:

GTA V is all about the attention to details, and lens flares are no exception:

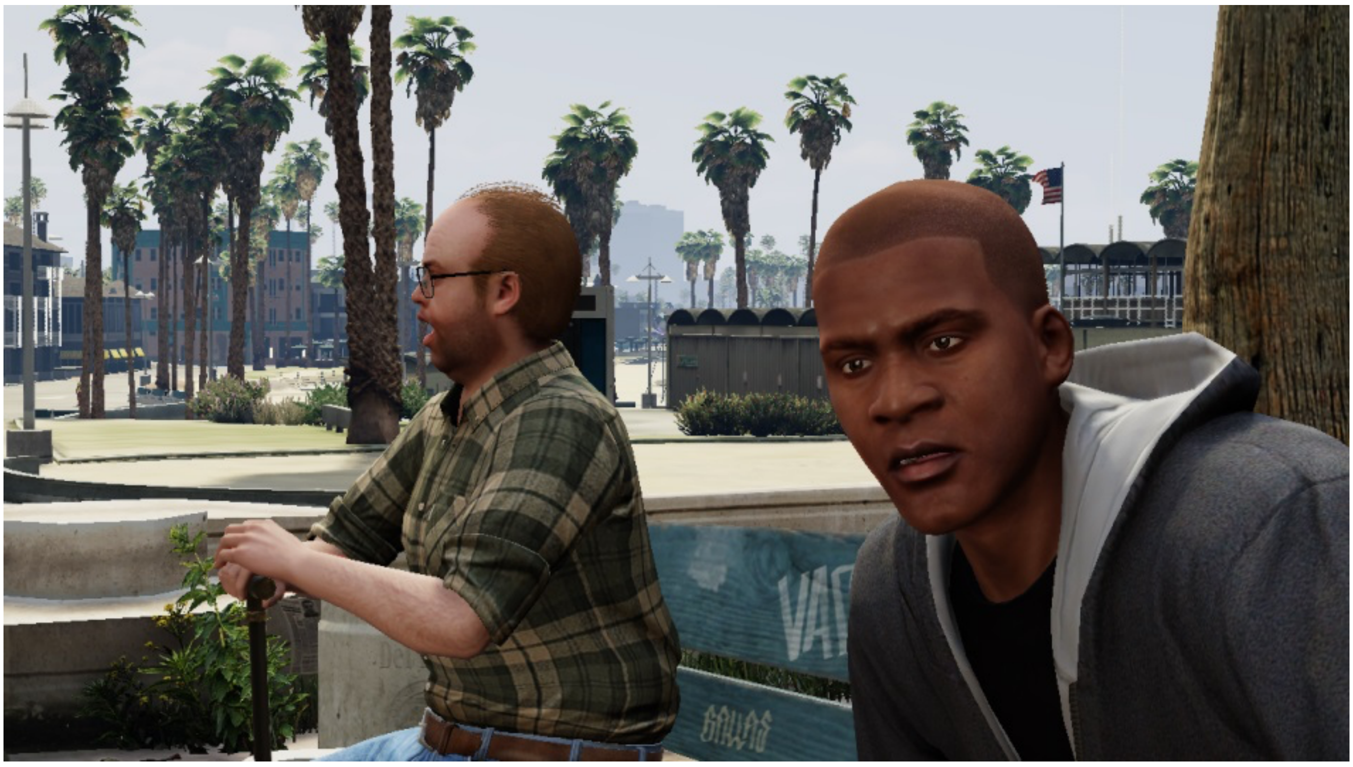
their size is proportional to the aperture of the camera. So if you suddenly look towards the sun, the lens flares are big at first, but then as the aperture narrows to lower the exposure the lens flares become smaller too. The animation below illustrates the phenomenon.
Another nice detail: if you switch to the first-person view, there are barely any lens-flares visible, because we are now seeing through human eyes, not through a camera anymore.

Anamorphic Lenses

Especially at night of for dark areas, the game simulates the artifacts of anamorphic lenses: long vertical or horizontal streaks, usually blue. Light streaks resulting from anamorphic lenses got really popular recently, with Hollywood (ab)using them in the latest science-fiction movies.
Here the effect is achieved using sprites, exactly like for the sun rays we saw previously, it is applied only on very bright sources like car headlights directly facing the camera.

Depth of Field

The scene below would look a little bit "artificial" if you were to see it in a movie, everything appears very sharp and crisp whereas in a movie you would expect the scenery in the background to be out-of-focus and look blurry.



Base Image

This is what the Depth of Field effect (DoF) is supposed to solve, by blurring out the areas of the image which are not in focus.
How is it applied? First a Circle of Confusion map (CoC map) is generated from the depth buffer. This tells us how much "out-of-focus" each pixel is, in other words how much "blurring" a pixel should receive. The CoC value of a pixel is solely dependent on its distance from the camera (so its depth) and the camera lens parameters.

Note though that GTA V stores the CoC as a signed value, meaning it varies between -1 and 1. The sign simply tells if the pixel is in front of or behind the in-focus area. For example pixels very far away and out-of-focus have a CoC of 1, pixels very close to the camera, out-of-focus, have a value of -1. Any value close to 0 means very little to no blurring.

Why is a signed value needed?
Because creating a good DoF is tricky, there are many cases to consider when you apply a "blur" to your scene.
For example you don't want a pixel out-of-focus in the background to bleed into a pixel in-focus in front of it. But now if you consider an out-of-focus pixel in the foreground, it's the opposite: you want it to bleed into the in-

focus pixels located behind it. So when it comes to blurring a pixel, it's not simply a question of "how much to blur", you also need to take into account whether the neighbor pixels are in-front or behind, in-focus or out-of-focus… Hence the signed value to be able to distinguish the different cases.
In the CoC map below I used the green channel for CoC > 0 and the red channel for CoC < 0 so you can visualize better.

Depth Map		**CoC Map**

So by the look of the CoC map we can know Lester in black is supposed to be in-focus, while Franklin in the front and the scenery in the back should be out-of-focus.

Then the engine extracts just the "front out-of-focus area": all the texels with a CoC < 0. This front CoC map is then blurred by a compute shader, using an horizontal pass followed by a vertical one.

What's the reason behind this? Well, Franklin's pixels have a CoC strength of about 0.7, while the bench right behind him is in focus with a CoC of 0. Now when the blur is performed, Franklin will look really blurry, but the bench will remain very sharp, the area near Franklin's right arm will look weird: you will have an abrupt change from strongly blurred arms pixel to suddenly sharp bench pixels. A hard silhouette quite easy to notice… This should not happen, the silhouette should be smooth, Franklin's pixels should bleed into the surrounding pixels.
This is what this blurred CoC map is supposed to achieve: smooth out the CoC map discontinuities to enable the bleeding of the "front out-of-focus area".

Now we have all we need to compute the depth of field. Historically blurs are computed at a lower resolution by a pixel shader, separated in a horizontal and vertical pass to optimize the computation. Here GTA V keeps the 2-pass approach but works at the original resolution and, to avoid crushing the performance, use a compute shader instead of a pixel shader. It makes sense because compute shaders can be particularly adapted for computing large-kernel blurs. When computing the final “blurred” value of a pixel, the color will be influenced by a smaller or larger number of neighbor pixels (depending on the pixel’s CoC) and certain neighbors might be excluded if they lead to incorrect bleeding.

Base	Depth	CoC	Blurred Front CoC

2-pass blur through compute shader

This effect gives a whole other dimension to the frame, like in a movie when the director would keep in focus a character to draw attention to him.

Conclusion

There are still many other post-processing effects we could talk about but this series of posts about GTA V has already become way longer than I originally planned.
There are heat haze, god rays (sometimes in screen-space by making bright areas bleed, other times added manually inside the scene as meshes), or motion blur (seems to be a hybrid approach, with a special pass doing a directional blur based only on the camera rotation direction excluding the player pixels thanks the stencil buffer acting like a mask).
The “Wasted” screen when your character dies is also pure post-process: after the scene is rendered normally it is blurred, turned into grayscale, then vignetting and film grain are added and finally the text is drawn on

top of it.

Well I hope I could shed some light on how the very secretive Rockstar managed to create a title considered by many as a landmark in the video game history. Its vast universe, its immersion and attention to details plus the fact Rockstar managed to make it run on the old generation of consoles make GTA V a really amazing title.

Links

- *The tech that built an empire: how Rockstar created the world of GTA 5* featuring an interview of Aaron Garbut.
- *GTA V NVIDIA Performance Guide* with details about the different graphics settings.
- *Renderdoc* which made picking into GTA V internals a breeze.

More discussion on this very topic: Slashdot, Hacker News, Reddit, John Carmack on Twitter.

GTA V - Graphics Study - Part 3

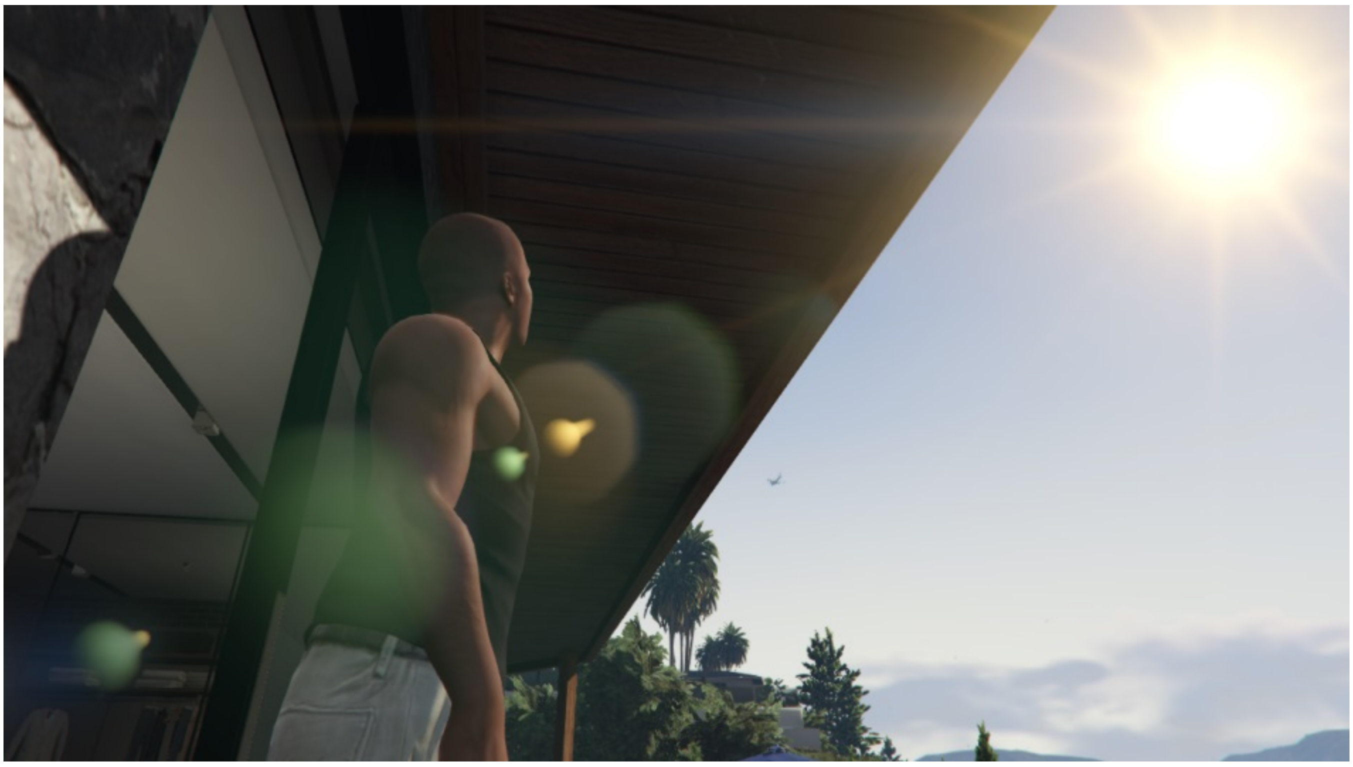
Posted by Adrian Courrèges   Nov 2nd, 2015

Post Processing Effects

Post-effects are performed once the scene has been rendered, to enhance, fix artifacts, change the mood…
We saw in *Part 1* how a few post-effects were applied, like bloom, anti-aliasing or tone-mapping. But there are several other effects used in GTA V.

Lens Flares & Light Streaks

When the light goes through a real-world lens, the scattering and internal reflections sometimes cause artifacts.
What I call "lens flares" here is a collection of bright spots along the axis defined by a bright light source and the center of the screen. There are also "light streaks" which are rays originating from the light source. These artifacts are very common in movies, when they are added to a game frame they can give a kind of "cinematic" feeling.

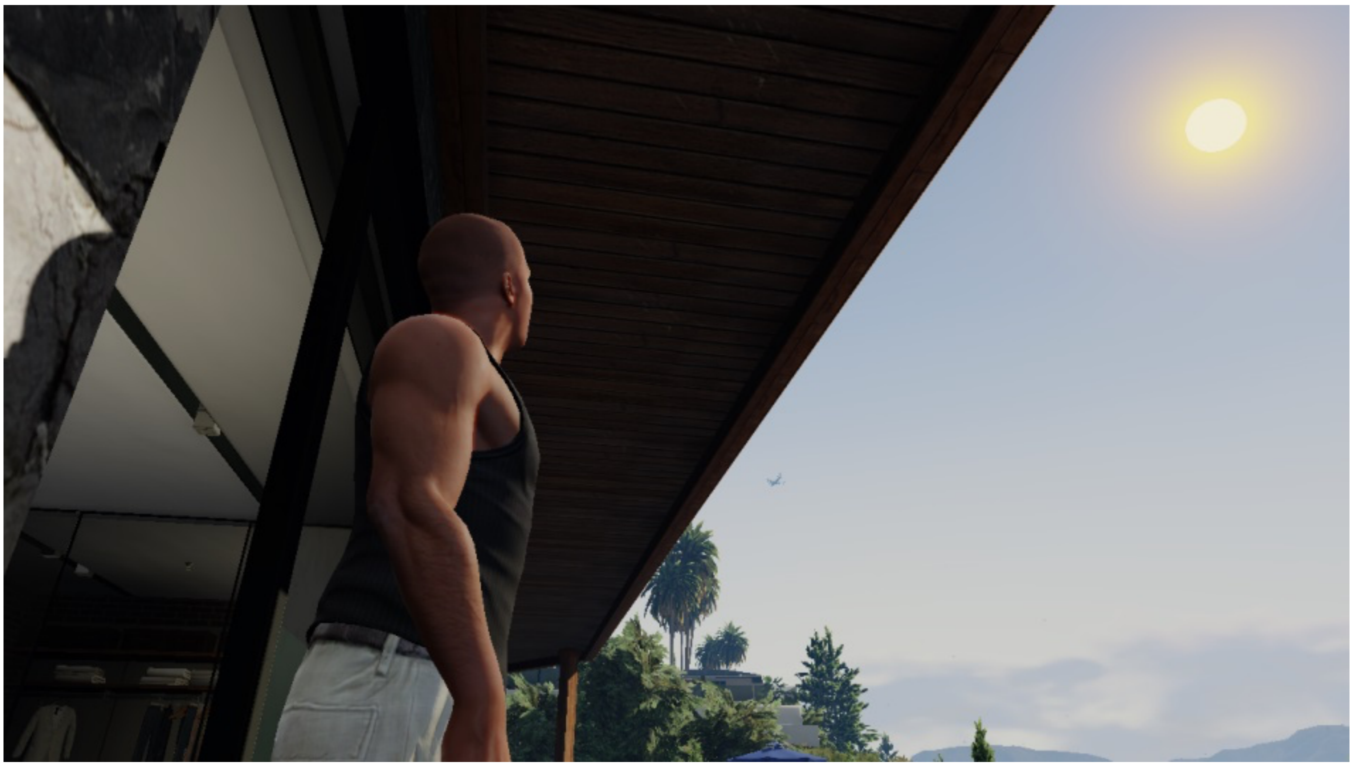


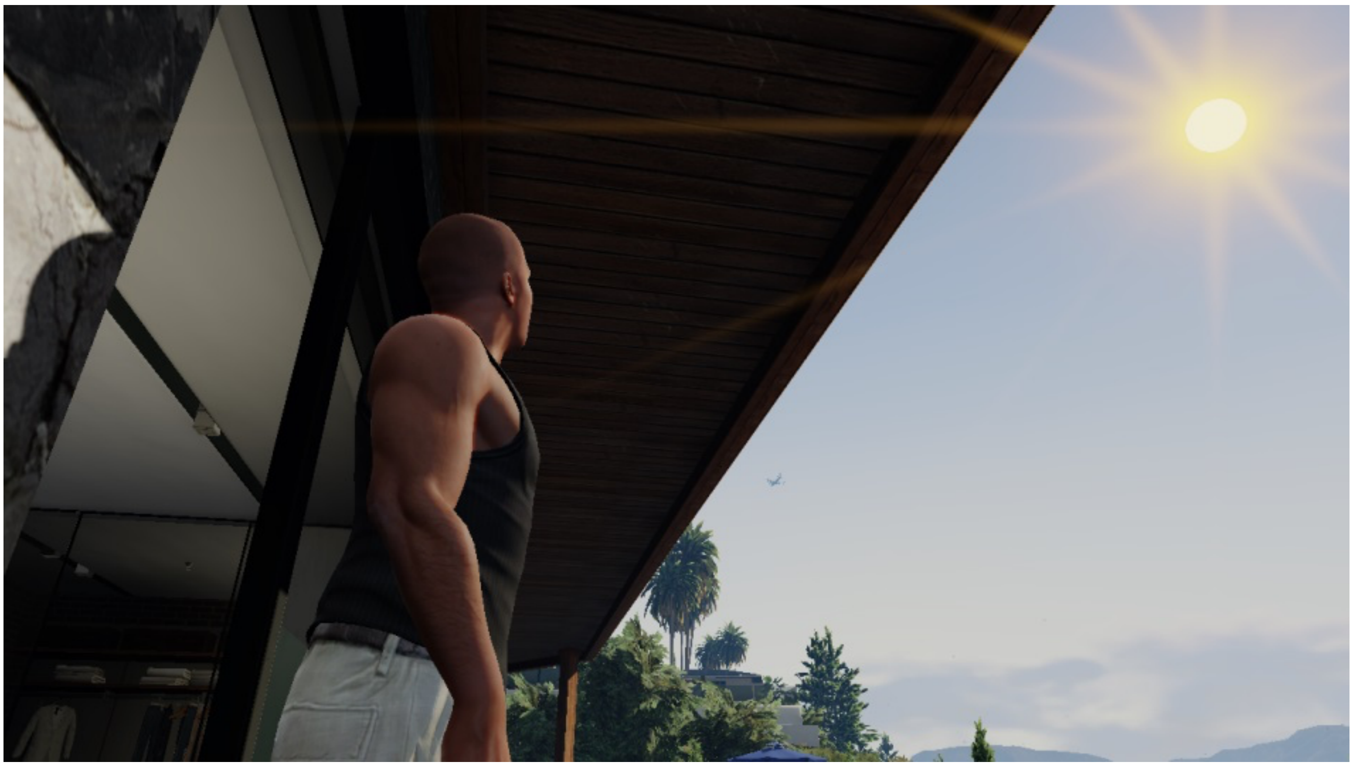
Lens Flares & Light Streaks

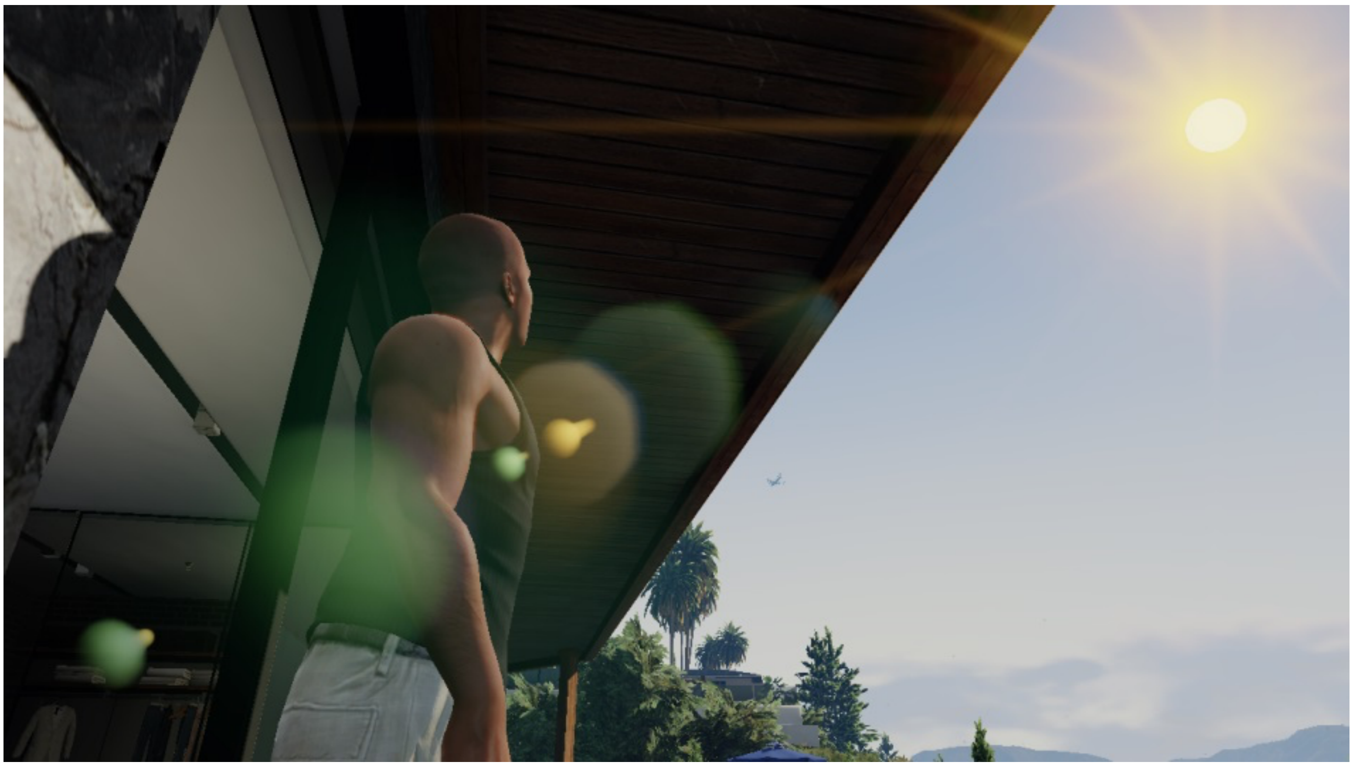
There are usually 2 ways to render such artifacts:

- *image-based*: extracting the brightest areas, duplicating and deforming them. Works for any number of bright light sources.
- *sprite-based*: adding textured-sprites and managing their positions manually. Each light source must be handled separately but artists can have more control over the artifact shape, color, intensity…

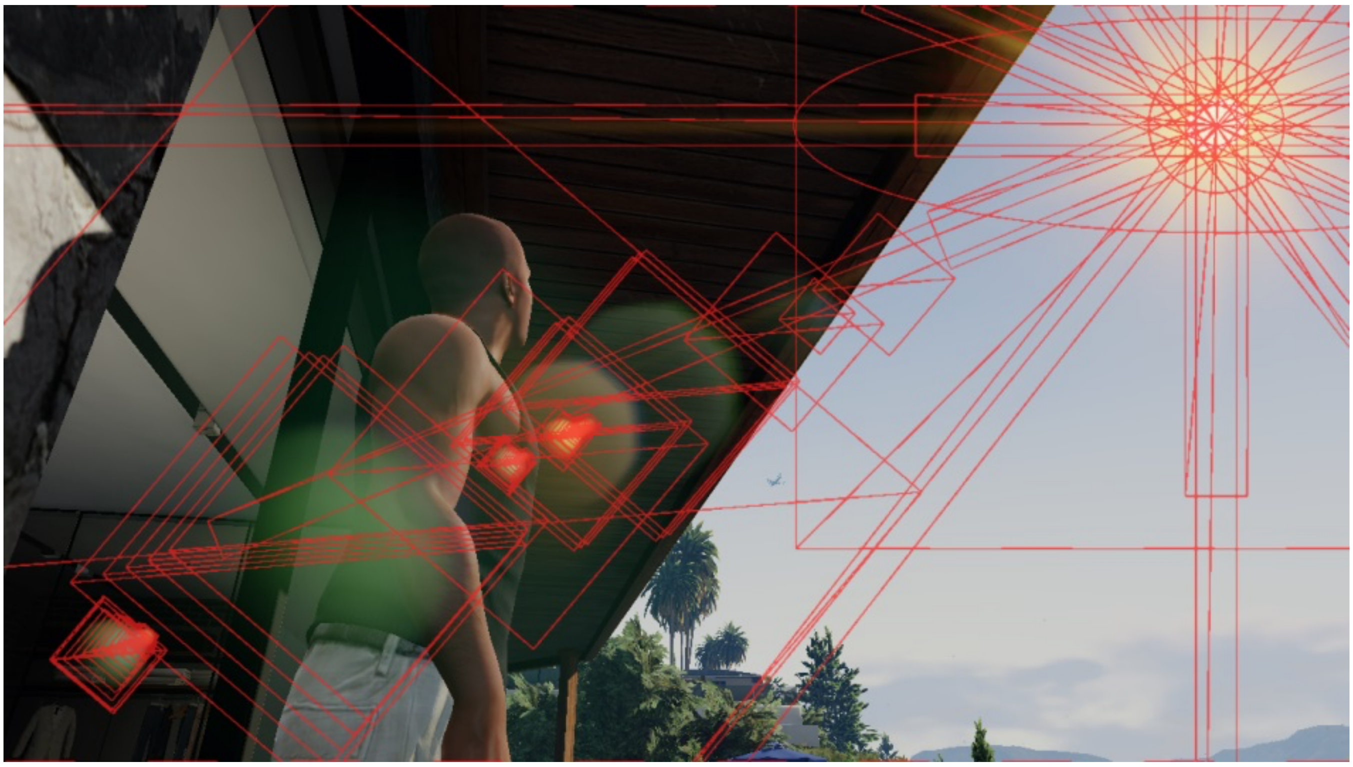
GTA V actually uses both techniques: the image-based approach is used to add a subtle blue halo at the bottom-left corner of the image, it's actually a symmetric of the bright-pass buffer. But the most visible artifacts in this scene result from the sprite-based approach, it is applied for the sun only. First, light streaks are added by rendering 12 rotated quads centered around the sun. Then for the lens flares, 70 sprites are drawn along the axis "sun – screen center". Artifacts come closer to each other as the camera points towards the sun.





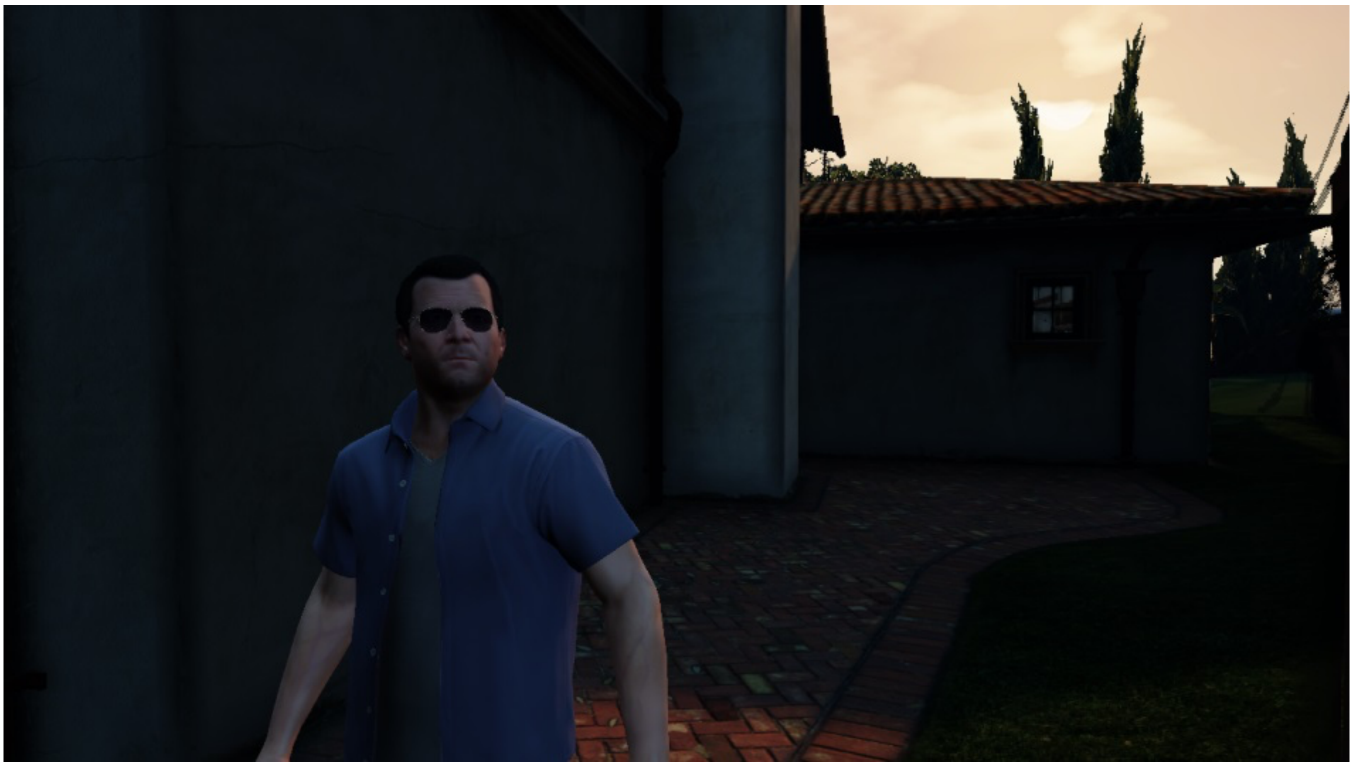


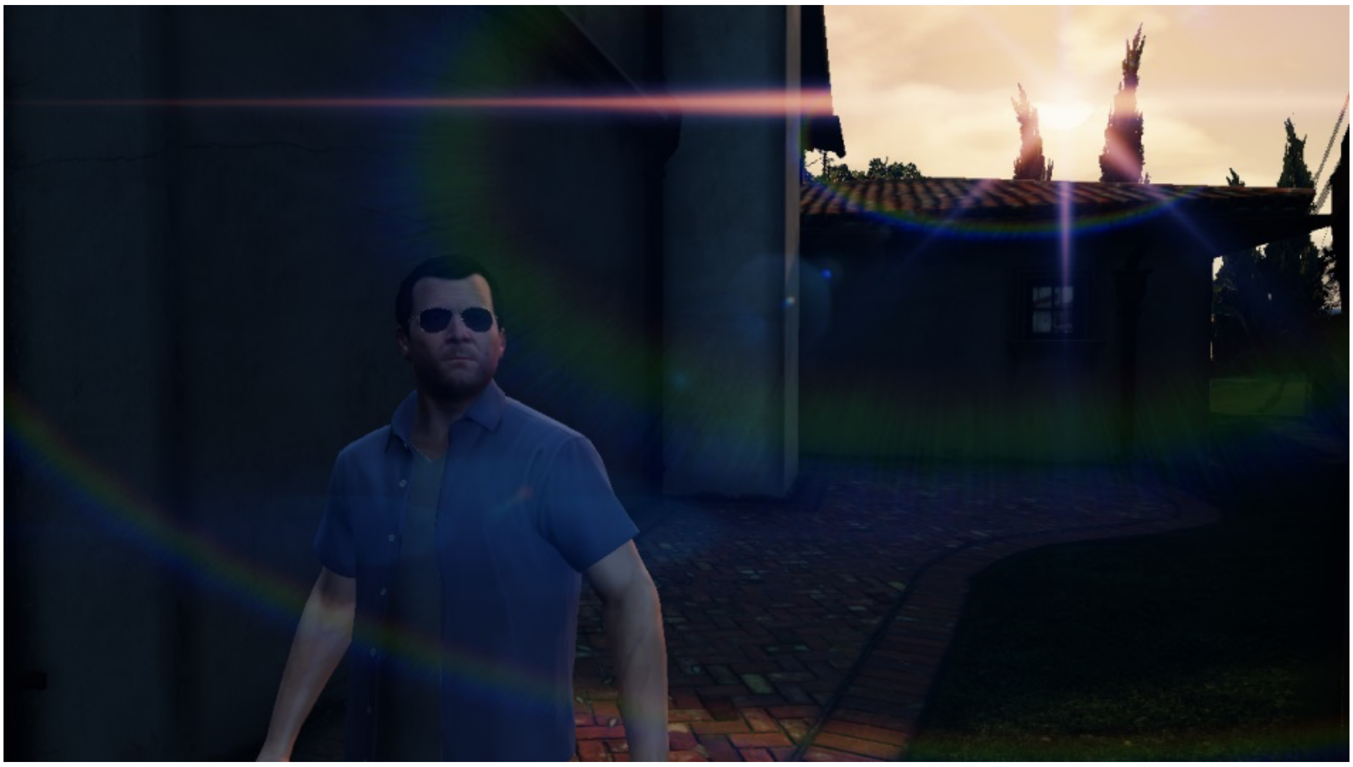
Base + Light Streaks + Lens Flares

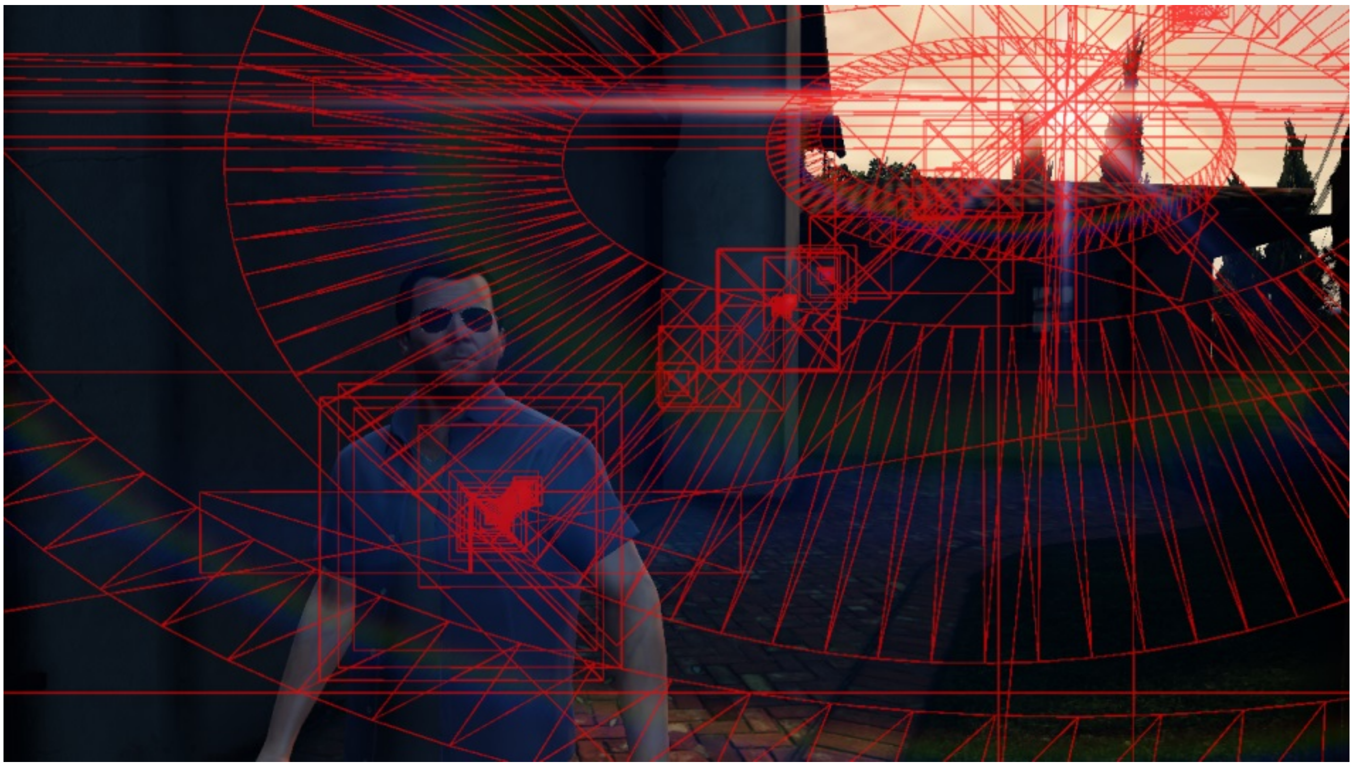


Base + Light Streaks + Lens Flares (Wireframe Highlight)

There are several sprites the engine uses to simulate different lens artifacts:

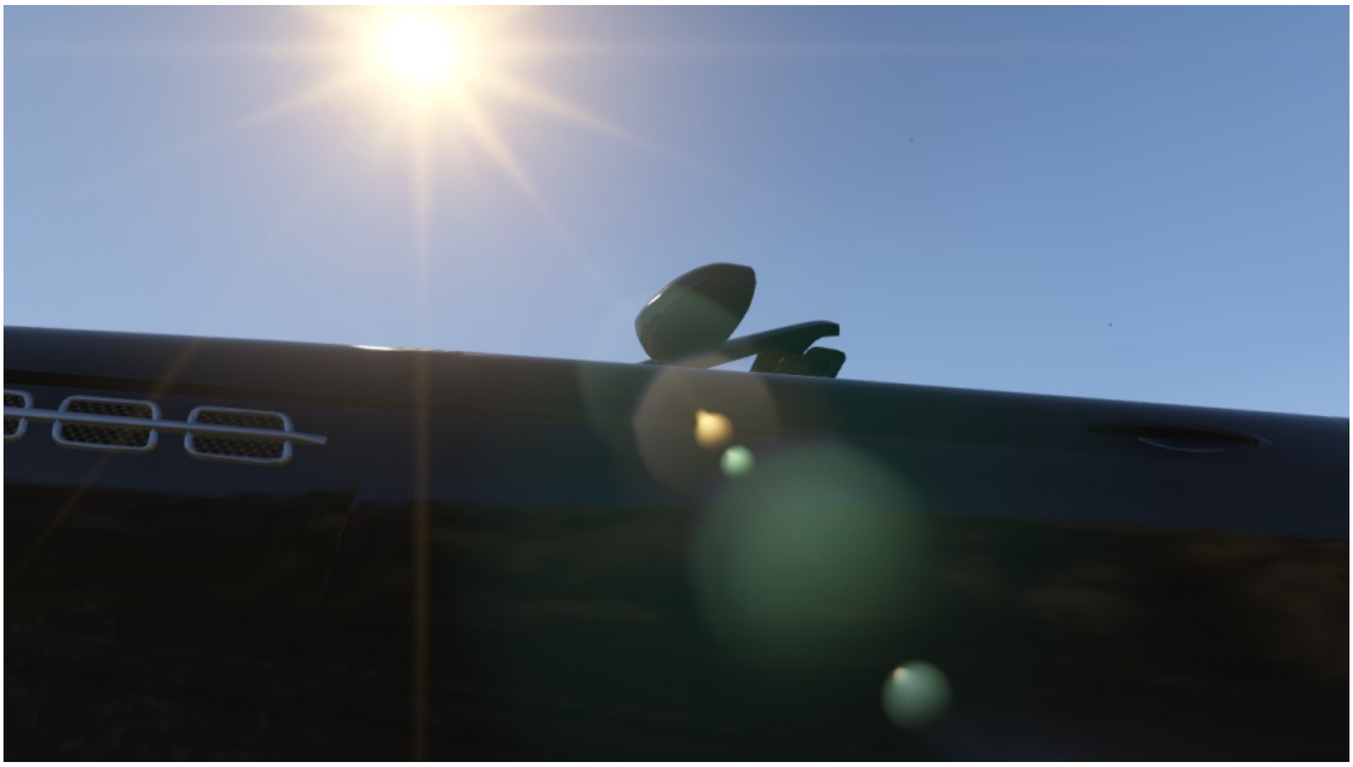
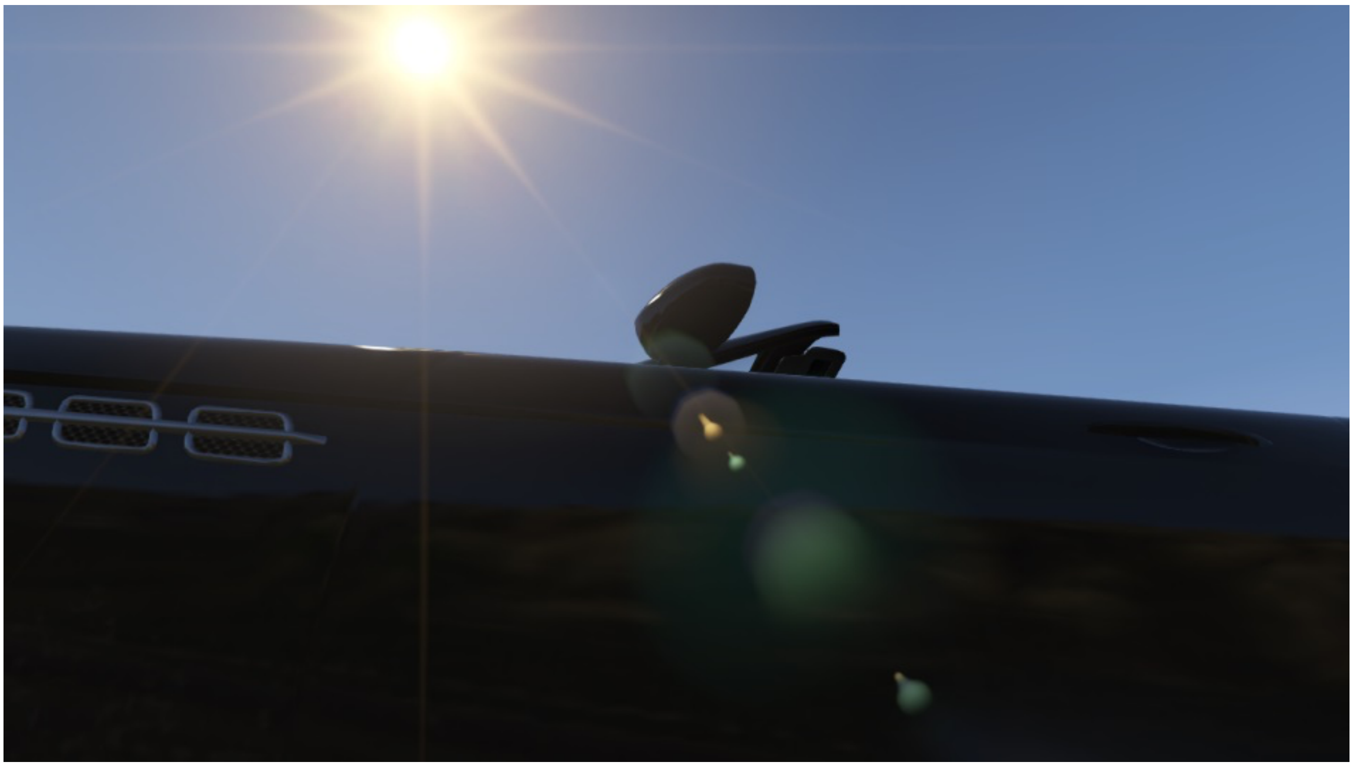






Base + Flares (Wireframe)

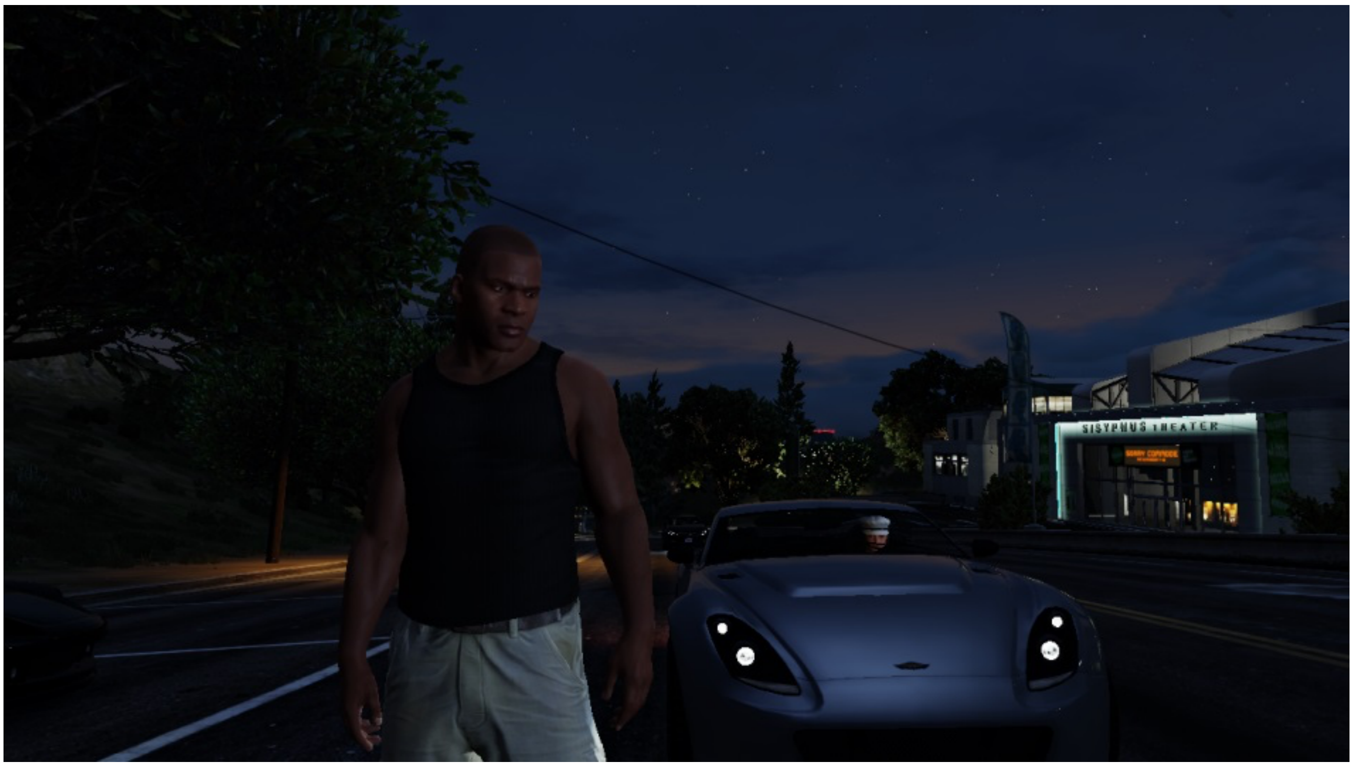
GTA V is all about the attention to details, and lens flares are no exception: their size is proportional to the aperture of the camera. So if you suddenly look towards the sun, the lens flares are big at first, but then as the aperture narrows to lower the exposure the lens flares become smaller too. The animation below illustrates the phenomenon.
Another nice detail: if you switch to the first-person view, there are barely any lens-flares visible, because we are now seeing through human eyes, not through a camera anymore.

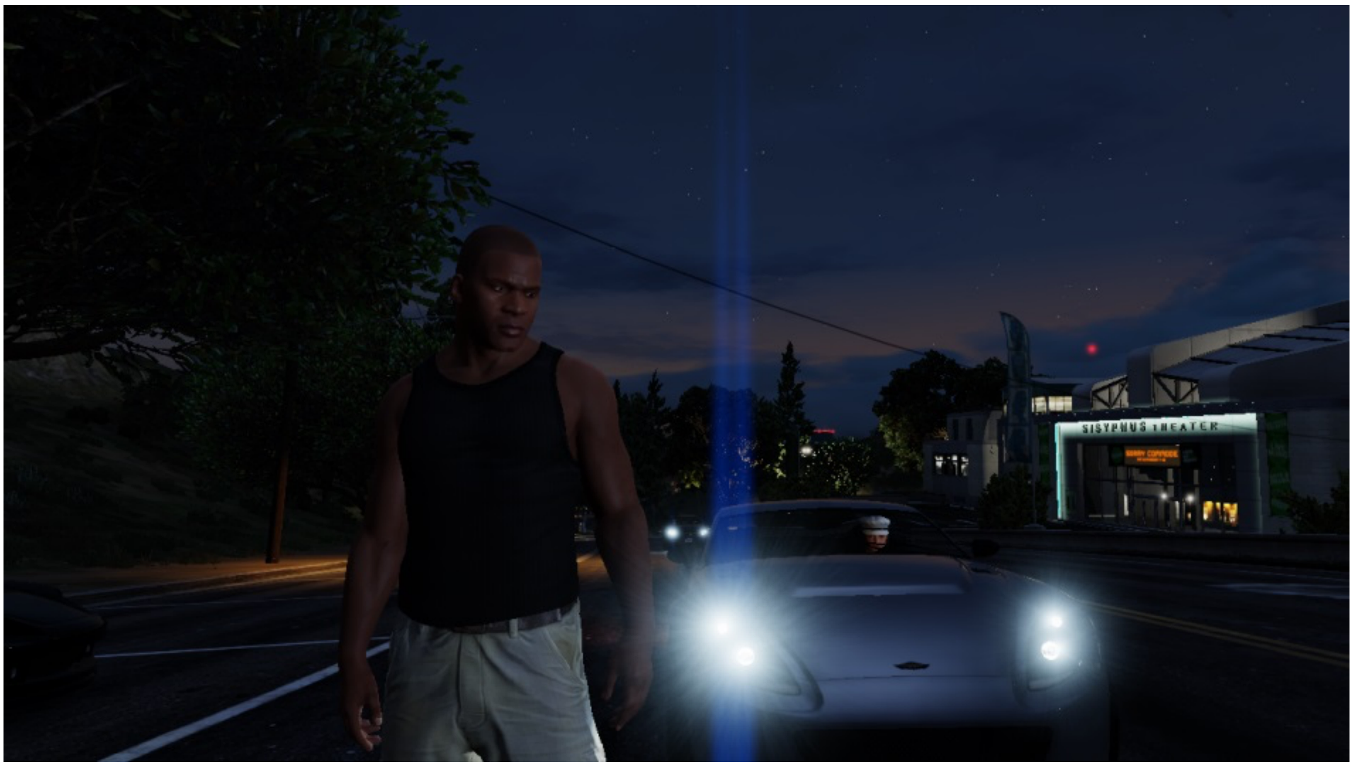



Anamorphic Lenses

Especially at night of for dark areas, the game simulates the artifacts of anamorphic lenses: long vertical or horizontal streaks, usually blue. Light streaks resulting from anamorphic lenses got really popular recently, with Hollywood (ab)using them in the latest science-fiction movies.
Here the effect is achieved using sprites, exactly like for the sun rays we saw previously, it is applied only on very bright sources like car headlights

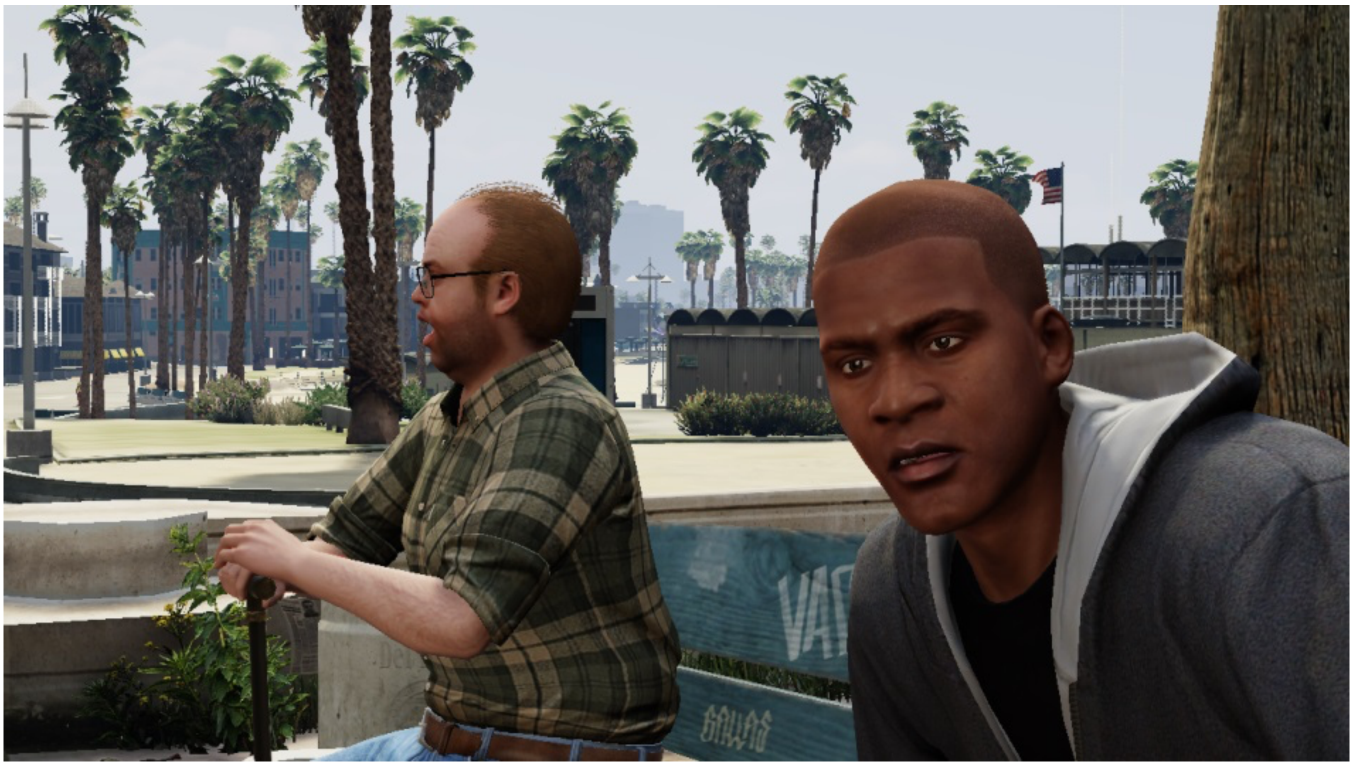
directly facing the camera.





Depth of Field

The scene below would look a little bit “artificial” if you were to see it in a movie, everything appears very sharp and crisp whereas in a movie you would expect the scenery in the background to be out-of-focus and look blurry.



Base Image

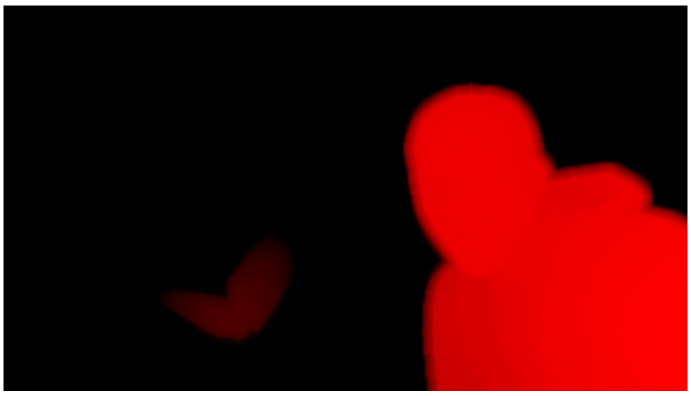
This is what the Depth of Field effect (DoF) is supposed to solve, by blurring out the areas of the image which are not in focus.
How is it applied? First a Circle of Confusion map (CoC map) is generated from the depth buffer. This tells us how much "out-of-focus" each pixel is, in other words how much "blurring" a pixel should receive. The CoC value of a pixel is solely dependent on its distance from the camera (so its depth) and the camera lens parameters.

Note though that GTA V stores the CoC as a signed value, meaning it varies between -1 and 1. The sign simply tells if the pixel is in front of or behind the in-focus area. For example pixels very far away and out-of-focus have a CoC of 1, pixels very close to the camera, out-of-focus, have a value of -1. Any value close to 0 means very little to no blurring.

Why is a signed value needed?
Because creating a good DoF is tricky, there are many cases to consider when you apply a "blur" to your scene.
For example you don't want a pixel out-of-focus in the background to bleed into a pixel in-focus in front of it. But now if you consider an out-of-focus pixel in the foreground, it's the opposite: you want it to bleed into the in-

focus pixels located behind it. So when it comes to blurring a pixel, it's not simply a question of "how much to blur", you also need to take into account whether the neighbor pixels are in-front or behind, in-focus or out-of-focus… Hence the signed value to be able to distinguish the different cases.
In the CoC map below I used the green channel for CoC > 0 and the red channel for CoC < 0 so you can visualize better.

So by the look of the CoC map we can know Lester in black is supposed to be in-focus, while Franklin in the front and the scenery in the back should be out-of-focus.



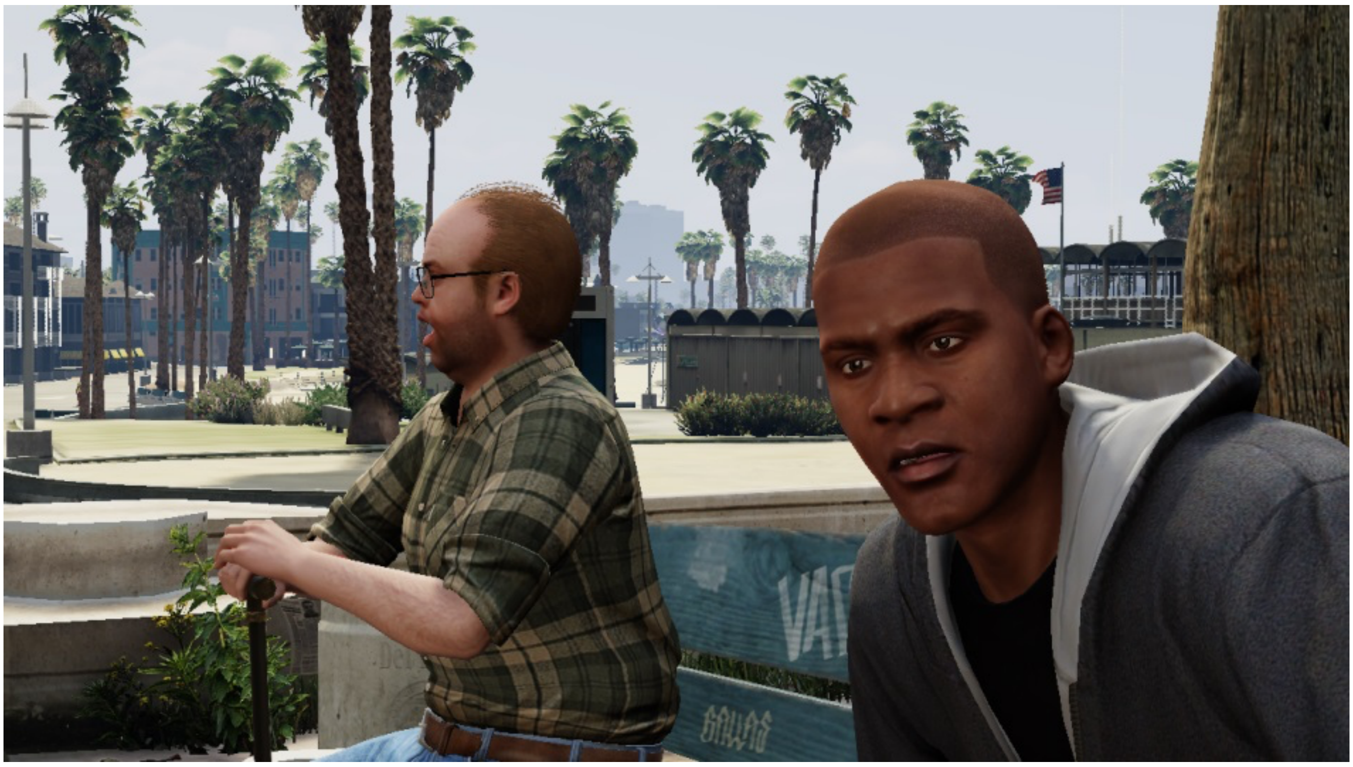
Then the engine extracts just the "front out-of-focus area": all the texels with a CoC < 0. This front CoC map is then blurred by a compute shader, using an horizontal pass followed by a vertical one.

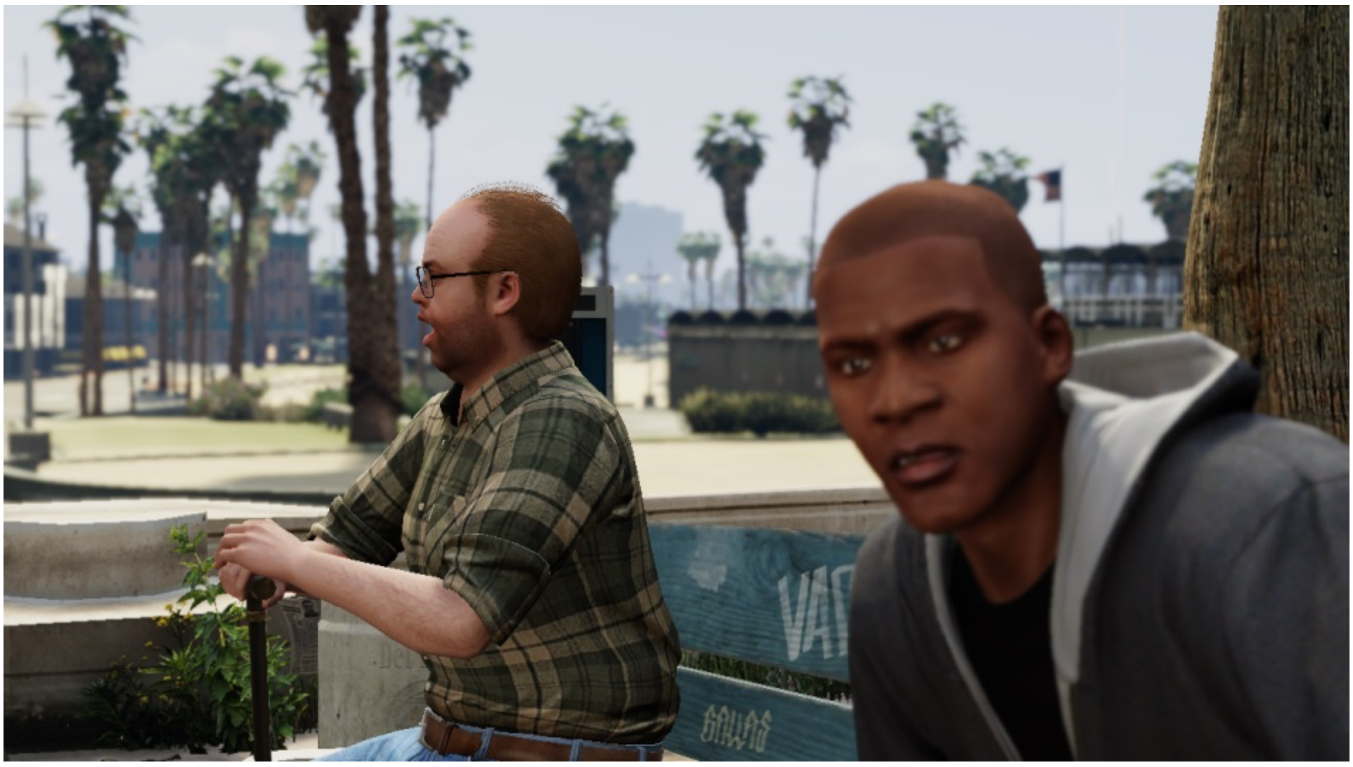
What's the reason behind this? Well, Franklin's pixels have a CoC strength of about 0.7, while the bench right behind him is in focus with a CoC of 0. Now when the blur is performed, Franklin will look really blurry, but the bench will remain very sharp, the area near Franklin's right arm will look weird: you will have an abrupt change from strongly blurred arms pixel to suddenly sharp bench pixels. A hard silhouette quite easy to notice… This should not happen, the silhouette should be smooth, Franklin's pixels should bleed into the surrounding pixels.
This is what this blurred CoC map is supposed to achieve: smooth out the CoC map discontinuities to enable the bleeding of the "front out-of-focus area".

Now we have all we need to compute the depth of field. Historically blurs are computed at a lower resolution by a pixel shader, separated in a horizontal and vertical pass to optimize the computation. Here GTA V keeps the 2-pass approach but works at the original resolution and, to avoid crushing the performance, use a compute shader instead of a pixel shader. It makes sense because compute shaders can be particularly adapted for computing large-kernel blurs. When computing the final “blurred” value of a pixel, the color will be influenced by a smaller or larger number of neighbor pixels (depending on the pixel’s CoC) and certain neighbors might be excluded if they lead to incorrect bleeding.

Base	**Depth**	**CoC**	**Blurred Front CoC**

2-pass blur through compute shader





This effect gives a whole other dimension to the frame, like in a movie when the director would keep in focus a character to draw attention to him.

Conclusion

There are still many other post-processing effects we could talk about but this series of posts about GTA V has already become way longer than I originally planned.
There are heat haze, god rays (sometimes in screen-space by making bright areas bleed, other times added manually inside the scene as meshes), or motion blur (seems to be a hybrid approach, with a special pass doing a directional blur based only on the camera rotation direction excluding the player pixels thanks the stencil buffer acting like a mask).
The “Wasted” screen when your character dies is also pure post-process: after the scene is rendered normally it is blurred, turned into grayscale, then vignetting and film grain are added and finally the text is drawn on top of it.

Well I hope I could shed some light on how the very secretive Rockstar managed to create a title considered by many as a landmark in the video game history. Its vast universe, its immersion and attention to details plus the fact Rockstar managed to make it run on the old generation of consoles make GTA V a really amazing title.

Links

- *The tech that built an empire: how Rockstar created the world of GTA 5* featuring an interview of Aaron Garbut.
- *GTA V NVIDIA Performance Guide* with details about the different graphics settings.
- *Renderdoc* which made picking into GTA V internals a breeze.

More discussion on this very topic: Slashdot, Hacker News, Reddit, John Carmack on Twitter.