Práctica 1 – Real Time

(Asignación 24/03; Entrega 21/04 a las 23:59)

¿Cómo entregar la práctica?

Se habilitará una entrega por Aula Virtual. Se deberá subir un fichero zip que incluirá los ficheros modificados y elementos de testing utilizados para el desarrollo de la práctica.

Especificaciones generales:

En esta práctica se ha de programar características usadas en motores de tiempo real.

Esta práctica consta de dos partes, una parte guiada, y otra a elección del alumno. Además se provee de una implementación básica de un motor "deferred" que renderiza materiales lambertianos en dos pasadas: gbuffer, y composición.

El alumno puede modificar las diferentes partes del motor según sus necesidades y diseños.

Instalación:

Requisitos:

- Instalar *Visual Studio 2019*
- Instalar última versión de *CMake: https://cmake.org/*
- Instalar última versión de *Vulkan: https://www.vulkan.org/*

Pasos:

1. Se proporciona un script en el directorio raíz de la práctica llamado "*gen_prj.cmd*" que descarga las dependencias necesarias.
2. En el directorio ./shaders se encuentra un script que compila los shaders necesarios:
    a. Editar el script compile.bat. Modificar el *path* al ejecutable glslc.exe.
    b. Ejecutar el script
3. Ejecutar la solución.

Si la instalación se ha realizado correctamente, se debería visualizar la siguiente escena:

Material PBR

El código base proporciona la implementación de un material lambertiano básico. En este apartado vamos a implementar un material *PBR* similar a los que podemos encontrar en motores comerciales. En nuestro caso vamos a seguir la especificación de *Unreal*.

En primer lugar vamos a modificar la pasada *DefferedPassVk* para incluir el shader responsable de renderizar los materiales *PBR*.

1. En este caso vamos a necesitar definir un *VkPipeline* nuevo con la misma configuración que los materiales Lambertianos, pero sustituyendo el *shader* de fragmentos a uno nuevo.

2. En el nuevo *shader* vamos a escribir en el *GBuffer* los parámetros de cada objeto correspondientemente:
    a. Escribimos el id de material al *attachment* correspondiente
    b. Escribimos el resto de parámetros al *GBuffer*
3. En la etapa de composición necesitamos comprobar el id del material, y dependiendo del id aplicamos uno u otro modelo de iluminación

En este motor vamos a implementar una *BRDF* en la cual la parte difusa la modelamos mediante *Lambert*, y la parte especular mediante microfacetas, usando el modelo de microfacetas especular de *Cook-Torrance*.

$$f(\mathbf{l},\mathbf{v}) = \frac{D(\mathbf{h})\,F(\mathbf{v},\mathbf{h})\,G(\mathbf{l},\mathbf{v},\mathbf{h})}{4\,(\mathbf{n}\cdot\mathbf{l})\,(\mathbf{n}\cdot\mathbf{v})}$$

En primer lugar, vamos a calcular la distribución de normales (*NDF*) mediante *GGX/Trowbridge-Reitz*, y el término de *roughness* será transformado a:

$$\alpha = Roughness^2.$$

$$D(\mathbf{h}) = \frac{\alpha^2}{\pi\,((\mathbf{n}\cdot\mathbf{h})^2\,(\alpha^2-1)+1)^2}$$

En segundo lugar, vamos a calcular el término *G*, es decir, el término geométrico. En este caso vamos a utilizar el modelo de *Schlick*, pero con $k = k = \alpha/2$. Antes de reparametrizar el *roughness* vamos a realizar el siguiente ajuste para luces analíticas:

$$k = \frac{(Roughness+1)^2}{8}$$

$$G_1(\mathbf{v}) = \frac{\mathbf{n}\cdot\mathbf{v}}{(\mathbf{n}\cdot\mathbf{v})(1-k)+k}$$

$$G(\mathbf{l},\mathbf{v},\mathbf{h}) = G_1(\mathbf{l})\,G_1(\mathbf{v})$$

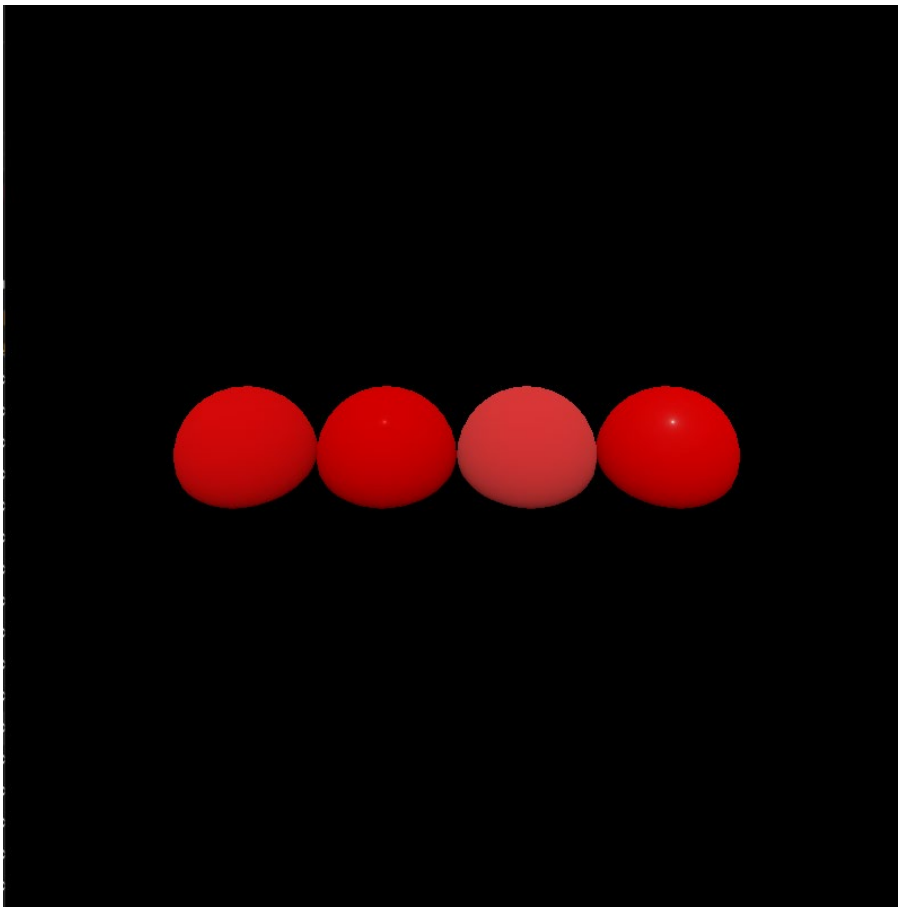
Por último, vamos a calcular el término de *Fresnel (F)*. En este caso vamos a utilizar la aproximación de *Schilick*, pero con una modificación: vamos a utilizar una *Gaussiana Esférica* para reemplazar la potencia:

$$F(\mathbf{v},\mathbf{h}) = F_0 + (1-F_0)\,2^{(-5.55473(\mathbf{v}\cdot\mathbf{h})-6.98316)(\mathbf{v}\cdot\mathbf{h})}$$

Donde *F0* es la reflectancia especular en la incidencia de la normal. Además, el parámetro *metallic* nos define si el material es dieléctrico o conductor. Es decir, en el caso de los metales el reflejo especular puede estar tintado. Normalmente se asume que el valor de *F0* para la mayoría de los materiales dieléctricos es 0.04. Por tanto, será necesario realizar una interpolación entre el albedo y el valor por defecto mediante el factor *metallic*.

NOTA: El término *Fresnel* puede ser sustituido por la aproximación de *Schlick* sin la *Gaussiana Esférica*.

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$



Prepasada de profundidad

En motores *Real Time* es común utilizar una prepasada de renderizado de solo profundidad para descartar fragmentos en futuras pasadas y optimizar el tiempo de ejecución. En este apartado vamos a implementar esta pasada, para utilizarla posteriormente como depth buffer a la hora de generar el *GBuffer*.

Para realizar su implementación vamos a realizar los siguientes pasos:

1. En la clase *DeferredPassVK* vamos a necesitar cambiar la configuración del *depth buffer*:
    a. Modificar la configuración del attachment correspondiente al *buffer* de profundidad a:

```
attachments[ 4 ].initialLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
attachments[ 4 ].finalLayout    = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

b. Cambiar la configuración de la estructura a:

```
VkPipelineDepthStencilStateCreateInfo depth_stencil{};
depth_stencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depth_stencil.depthTestEnable = VK_TRUE;
depth_stencil.depthWriteEnable = VK_FALSE;
depth_stencil.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
depth_stencil.depthBoundsTestEnable = VK_FALSE;
depth_stencil.stencilTestEnable = VK_FALSE;
depth_stencil.flags = 0;
```

c. En la función *VkCommandBuffer draw(const Frame& i_frame)* hay que eliminar la limpieza del *depth buffe*r:

```
VkClearValue clear_values[ 4 ];
clear_values[ 0 ].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
clear_values[ 0 ].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
clear_values[ 0 ].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
clear_values[ 0 ].color = { { 0.0f, 0.0f, 0.0f, 0.0f } };
render_pass_info.clearValueCount = 4;
render_pass_info.pClearValues = clear_values;
```

2. Vamos a necesitar crear una clase nueva que herede de la clase base *RenderPassVK*.
3. Esta clase va a necesitar implementar todas las funciones virtuales de la clase padre:
    a. *virtual bool initialize() = 0;*
    b. *virtual void shutdown() = 0;*
    c. *virtual VkCommandBuffer draw(const Frame&) = 0;*
    d. *virtual void addEntityToDraw(const EntityPtr i_entity);*
4. Podemos tomar como punto de partida el esqueleto de la clase *DeferredPassVK*, y realizar las siguientes modificaciones:
    a. En la función *bool initialize() override* vamos a eliminar el shader de fragmentos ya que en vulkan no es necesario esta etapa para escribir al *buffer* de profundidad.
    b. En la función *void createRenderPass();* vamos a modificar el número de *VkAttachmentDescription* a un único con la siguiente configuración:

```
std::array<VkAttachmentDescription, 1> attachments = {};
// Depth attachment
attachments[ 0 ].format = m_depth_output.m_format;
attachments[ 0 ].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[ 0 ].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[ 0 ].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
```

```
attachments[ 0 ].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[ 0 ].stencilStoreOp = VK_ATTACHMENT_STORE_OP_STORE;
attachments[ 0 ].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[ 0 ].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
VkAttachmentReference depth_reference = {};
depth_reference.attachment = 0;
depth_reference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

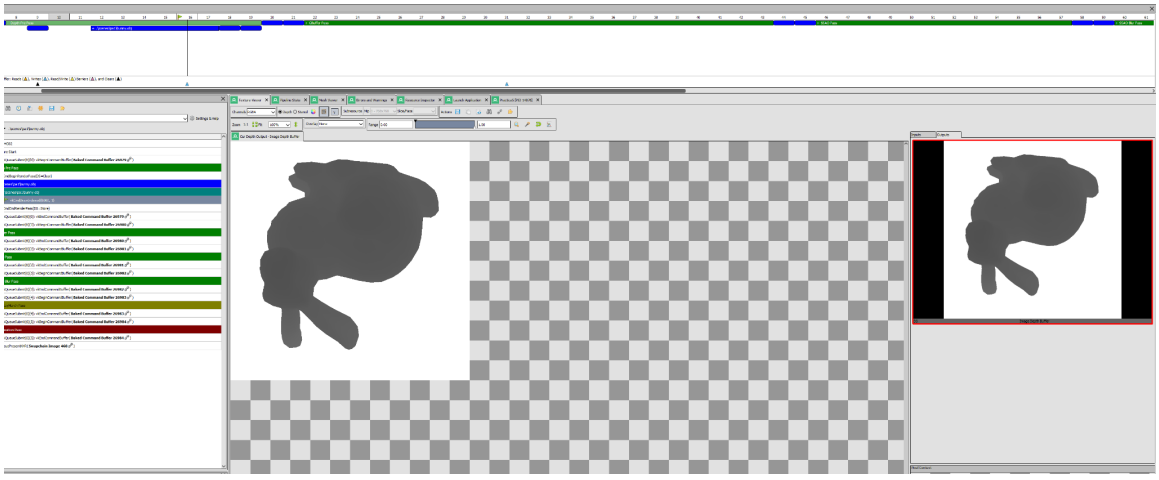
c. En la función *void createPipelines()* modificar la configuración del test de profundidad a:

```
VkPipelineDepthStencilStateCreateInfo depth_stencil{};
depth_stencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depth_stencil.depthTestEnable = VK_TRUE;
depth_stencil.depthWriteEnable = VK_TRUE;
depth_stencil.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
depth_stencil.depthBoundsTestEnable = VK_FALSE;
depth_stencil.stencilTestEnable = VK_FALSE;
depth_stencil.flags = 0;
```

d. En la función *VkCommandBuffer draw(const Frame& i_frame)* limpiar el buffer de profundidad:

```
VkClearValue clear_values[ 1 ];
clear_values[ 0 ].depthStencil = { 1.0f, 0 };
render_pass_info.clearValueCount = 1;
render_pass_info.pClearValues = clear_values;
```

Una vez implementado podemos comprobar con herramientas de debug que nuestra nueva pasada se está ejecutando correctamente.



Screen Space Ambient Occlusion

Screen Space Ambient Occlusion es una técnica que permite calcular la oclusión en espacio de pantalla optimizando su cálculo. Esta técnica se divide en tres partes: cálculo de la oclusión, emborronamiento de la oclusión, y aplicación de la oclusión. En este apartado vamos a implementar estas pasadas en nuestro motor

Cálculo de oclusión ambiental:

Como hemos visto en la teoría, el cálculo de *SSAO* necesita de una pasada de visibilidad, que básicamente ya hemos realizado al exportar al *GBuffer* las posiciones, normales y profundidad.

En este punto vamos a crear una nueva pasada como en el primer apartado en que vamos a usar como entrada los attachments mencionados anteriormente, y como salida, vamos a usar un attachment de un único canal *float*. (Ya está creado con anterioridad, *m_ssao_attachment*)

En esta pasada tenemos que realizar la siguiente configuración:

- Deshabilitar depth test
- Crear los *descriptors* y *descriptor sets* para nuestros inputs. Al menos, necesitaremos:
    - Buffer de parámetros globales
    - Textura de posición + profundidad
    - Textura de normales
    - Textura de números aleatorios
    - *Buffer* de *kernels*
- Configurar el *framebuffer* con un único attachment (*m_ssao_attachment*)
- Usar los shaders de esta pasada en la definición del pipeline
- Usar un plano para renderizar el mapa de salida

El siguiente paso es la configuración de los atributos auxiliares de ruido para rotar y muestrear en la hemisfera en espacio de pantalla. Para ello, en el esqueleto tenemos definidas las siguientes funciones:

- *UtilsVK::createImage* función para crear una textura en gpu
- *UtilsVK::createBuffer* función para crear un buffer en gpu

Dadas estas funciones vamos a necesitar crear dos vectores del tipo *Vector4f* a partir de una secuencia de números aleatorios:

- *Kernel*: Se recomienda un tamaño de 64. Este vector debe crear las muestras para realizar la comprobación contra la profundidad.
- *Noise*: Se recomienda un tamaño de 4 * 4. Crea vectores para rotar el sistema de coordenadas en espacio de pantalla y añadir aleatoriedad.

Una vez asignados los descriptores correspondientes podemos empezar a desarrollar nuestro shader de cálculo de oclusión ambiental. Para ello podemos seguir las transparencias de *SSAO* vistas en la última clase.

NOTA: Recordar que es importante que todos los elementos trabajen en el mismo espacio de coordenadas.

NOTA: Se valorará con 0.5 extra la implementación del *DSSAO*

Emborronamiento de la oclusión ambiental

Al igual que en el caso anterior vamos a crear una nueva pasada de render. En este caso solo tendremos como entrada el *attachment* de oclusión ambiental, y como salida un nuevo *attachment* del mismo tipo que el anterior.

Se recomienda usar la pasada anterior como plantilla y ajustar los *descriptors, descriptor sets y shaders*, ya que el resto usa la misma configuración.

Por último, en esta pasada tenemos que implementar un filtro de caja que emborrone el cálculo anterior para eliminar los patrones de ruidos y obtener un mapa plausible.

NOTA: Se pueden usar otro tipos de filtros.

Aplicación de la oclusión ambiental

Esta etapa la vamos a implementar en la pasada *CompositionVk*. En esta etapa vamos a muestrear el mapa de oclusión ambiental y aplicarlo a la iluminación siguiendo los siguientes pasos:

1. Configurar el objeto *VkPipeline* para añadir una textura más que será la obtenida de la pasada anterior. Vamos a necesitar modificar los descriptores
2. En el shader de composición vamos a añadir el binding al nuevo input en la etapa de fragmentos y vamos a realizar los siguientes pasos:
    a. Muestrear el factor de oclusión ambiental a partir de las coordenadas de pantalla
    b. Multiplicar el factor de oclusión ambiental por el resultado de muestrear las luces

Parte no guiada

En este apartado se deja a elección del alumno la implementación de una característica de los motores de real time. Algunas de las opciones son:

- *Bloom*
- *Screen Space Reflections*
- *Screen Space Refractions*

- *Shadow Mapping*
- Luces de Área
- Renderizado de volúmenes
- Armónicos Esféricos
- *Level of Detail*
- *Procedural Rendering*
- *Signal Distance Fields*

Asimismo, el alumno puede proponer características no incluidas en la lista siempre y cuando estén consensuadas con el profesorado.