UNIVERSITAT POLITÈCNICA DE VALÈNCIA



Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

UNIVERSITAT POLITÈCNICA DE VALÈNCIA, DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

MÁSTER EN INTELIGENCIA ARTIFICIAL, RECONOCIMIENTO DE FORMAS E IMAGEN DIGITAL

# Renderizado de funciones de distancia mediante Raymarching en Unity.

Trabajo Final de Máster – Curso 2018 - 2019

Tutor: Francisco José Abad Cerdá

Autor: Adrián Ciborro Montes

# Índice

# Lista de figuras

# Resumen

La computación gráfica es un campo en continuo desarrollo que cambia industrias como el cine, la animación o los videojuegos. La visualización y creación de recursos 3D ha evolucionado para adaptarse a las necesidades y restricciones del hardware disponible en cada momento. La representación de objetos más usada actualmente son las mallas de polígonos y se renderizan con algoritmos de rasterización o trazado de rayos. Sin embargo, existen otras representaciones como las funciones de distancia que permiten modelar de forma más realista objetos 3D. Para visualizarlas se utiliza el método de raymarching que consiste en marchar sobre rayos hasta encontrar las primitivas en la escena. En la actualidad esta técnica es computacionalmente costosa para aplicaciones en tiempo real, pero el campo avanza rápidamente y en unos años su uso estará más extendido tanto para la creación de recursos 3D como para su visualización.

---

La computació gràfica és un camp en continu desenvolupament que canvia indústries com el cinema, l'animació o els vídeojocs. La visualització i creació de recursos 3D ha evolucionat per adaptar-se a les necessitats i restriccions del hardware disponible en cada moment. La representació d'objectes més usada actualment són les malles de polígons i es renderitzen amb algoritmes de rasterización o traçat de raigs. No obstant això, hi ha altres representacions com les funcions de distància que permeten modelar de forma més realista objectes 3D. Per visualitzarlos s'utilitza el mètode de raymarching que consisteix a marxar sobre rajos fins a trobar les primitives en l'escena. En l'actualitat aquesta tècnica és computacionalment costosa per a aplicacions en temps real, però el camp avança ràpidament i en uns anys el seu ús estarà més estès tant per a la creació de recursos 3D com per a la seva visualització.

---

Computer graphics is a field in continuous development that changes industries such as film, animation or videogames. The visualization and creation of 3D resources has evolved to adapt to the needs and restrictions of the hardware available at that time. The most commonly used object representation is polygon meshes and are rendered with rasterization or ray tracing algorithms. However, there are other representations such as distance functions that allow more realistic modeling of 3D objects. To visualize them, the raymarching method is used, which consists of marching on rays until the primitives are found in the scene. Currently, this technique is computationally expensive for real-time applications, but the field is advancing rapidly and in a few years its use will be more widespread both for the creation of 3D resources and for its visualization.

# 1. Introducción

En computación gráfica existen numerosas técnicas de visualización y renderización de objetos, siendo las más típicas la rasterización y el trazado de rayos en escenas compuestas por mallas de triángulos. Esta representación tiene únicamente información sobre la superficie, y el cálculo para saber si un punto intersecta o se encuentra en el interior del objeto es computacionalmente costoso. El renderizado de volúmenes o renderizado volumétrico son un conjunto de técnicas que tratan con volúmenes en lugar de superficies.

Las **funciones de distancia** son expresiones analíticas que representan volúmenes. Las ventajas de esta representación frente a representaciones típicas como las mallas de triángulos son:

- Se pueden evaluar en cualquier punto para saber si este se encuentra dentro o fuera del volumen.
- No tienen error de aproximación: para cualquier punto del espacio se puede calcular la distancia al volumen que representan y por tanto se pueden escalar o hacer zoom sin perder detalle. Esto contrasta con las representaciones mediante triángulos donde la densidad de triángulos y la distancia a la cámara hace que objetos curvos puedan verse "poligonizados" a distancias cortas.
- La composición de primitivas mediante operadores da como resultado formas con un estilo más orgánico y natural. Estas operaciones permiten crear formas complejas que son difíciles de modelar usando triángulos.
- Proporcionan implícitamente información sobre la iluminación global evaluando puntos alrededor del objeto. Esto hace que la implementación de técnicas de iluminación y sombreado modernas sea más fácil que en otros algoritmos de renderizado como la rasterización o el trazado de rayos.

El algoritmo que se usa para renderizar las funciones de distancia se llama **raymarching**. Se trata de una técnica de renderizado que pertenece a la familia de los algoritmos de trazado de rayos. Esta técnica se basa en, dado un rayo, marchar a pasos en la dirección del rayo hasta encontrar un punto de intersección con la superficie del volumen. La diferencia principal por la que se usa este algoritmo frente al trazado de rayos clásico es que, dado un conjunto de funciones de distancia, llamado campo de distancia, se desconoce dónde intersecta el rayo, siendo necesaria una técnica iterativa que explore el espacio, mientras que en el trazado de rayos tradicional se calcula la intersección del rayo con los triángulos que forman la escena.

En este trabajo presentamos una implementación en Unity de un sistema de renderizado de funciones de distancia mediante raymarching. El sistema se ha implementado en un package de Unity que se puede añadir a proyectos propios. El sistema permite declarar primitivas y realizar operaciones que las transforman y mezclan para crear escenas complejas.

Para finalizar se ha evaluado el rendimiento del sistema de renderizado en 3 escenas con distinto número de primitivas y transformaciones. Con estas escenas se ha hecho un conjunto de experimentos donde se prueban los algoritmos de iluminación especular, sombras suaves y oclusión ambiental. A partir de estos resultados se han identificado problemas y propuesto soluciones para el trabajo futuro.

# 2. Estado del arte

## 2.1 Aplicaciones de las funciones de distancia

Las funciones de distancia y el algoritmo de renderizado raymarching son técnicas actualmente muy usada en demoscenes. Las demoscenes son una subcultura de arte por computador que se popularizó en la década de 1980. Consisten en producir demos donde se muestran las habilidades de programación, arte visual y música en entornos con pocos recursos computacionales, siendo las categorías más típicas 64k y 4k, donde el ejecutable del programa debe de ocupar menos de 65536 y 4096 bytes, respectivamente. Algunos de los eventos de demoscenes más conocidos son Revision [21], la demoparty más grande a nivel mundial que se celebra en Alemania, o NVSCene [22], demoparty que se da en el evento NVISION organizado por NVIDIA en California donde, además, se dan clases y charlas científicas sobre computación gráfica. Además, en Scene.org Awards [20] se nominan y premian a las mejores demoscenes de cada año.

*Figura 1: Demos Elevated y Cdak*

Muchas de estas demos se publican en Shadertoy [14], una web y herramienta de creación de shaders en WebGL. Fue creada por Pol Jeremias e Íñigo Quílez en enero del 2013 y tiene alrededor de 11 mil contribuciones y es referenciada en multitud de artículos científicos.

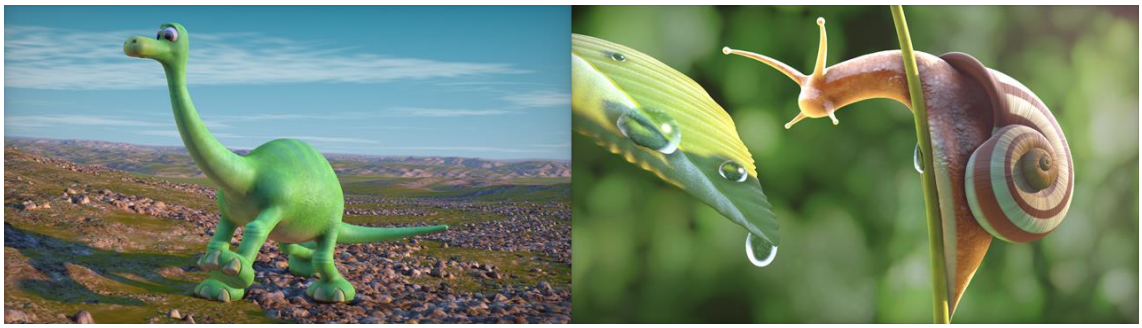


*Figura 2: Demos Arlo y Snail*

Las ventajas de usar funciones de distancia frente a mallas de triángulos en demoscenes son reducir el tamaño de los ejecutables para cumplir con los requisitos de cada categoría y calcular la iluminación global de la escena en tiempo real sin que sea necesario precalcularla y almacenarla en una textura, lo que supone también una reducción del tamaño de la demo.

En la edición del 2007 del SIGGRAPH [12], el congreso más importante a nivel mundial en informática gráfica, Chris Green de Valve presentó un método para renderizar glifos usando funciones de distancia en 2D [7]. Este método se basa en generar funciones de distancia a partir de texturas en alta resolución de los caracteres que se quiere renderizar. Puede usarse en dispositivos con hardware gráfico de gama baja, ya que no requiere el uso de shaders, tan sólo es necesario la funcionalidad de alpha-testing. Además, si el dispositivo soporta shaders programables esta técnica puede usarse en conjunto con otras para añadir efectos gráficos como el suavizado de bordes, contornos, iluminación o sombras.

*Figura 3: Comparativa filtrado bilineal, alpha testing y funciones de distancia*

Actualmente este método se usa en motores de videojuegos como Unity [15]. El plugin TextMesh Pro [18] tiene un conjunto de shaders que usan campos de distancia para renderizar las fuentes y permiten añadir efectos gráficos al texto.

En la edición del 2015 del SIGGRAPH, Daniel Wright de Epic Games presentó un nuevo método para calcular la oclusión ambiental de una escena mediante campos de distancia llamado Distance Field Ambient Occlusion (DFAO) [8]. Este método está actualmente implementado en Unreal Engine 4 [16] y se basa en precalcular y generar por cada objeto rígido un campo de distancias trazando rayos. El algoritmo se ejecuta principalmente en la GPU y está optimizado para usarse en hardware de gama media.

*Figura 4: Oclusión ambiental usando campos de distancia en Unreal Engine 4*

El 31 de agosto de 2018 se lanzó el videojuego Claybook [17], un juego de simulación de arcilla con un entorno totalmente destructible. En la Game Developers Conference [13] de 2018, Sebastian Aaltonen, presentó la implementación y detalles técnicos del videojuego [9]. Tanto los personajes como el mundo están modelados usando campos de distancia. La física y simulación de fluidos también está implementada usando campos de distancia. La iluminación, sombras y oclusión ambiental se calculan en tiempo real. Los objetos están representados mediante campos de distancias precalculados en texturas volumétricas (texturas 3D) usando interpolación trilineal y se utilizan distintos niveles de mip maps de estas texturas según la distancia de la cámara. Esta representación de los objetos la llaman pinceles y se agrupan en un atlas, permitiendo que el número de objetos escale proporcionalmente con la complejidad de la escena. La resolución de cada pincel se encuentra entre [$32^3$, $128^3$] o [32kB, 2MB], dependiendo de la complejidad del pincel.

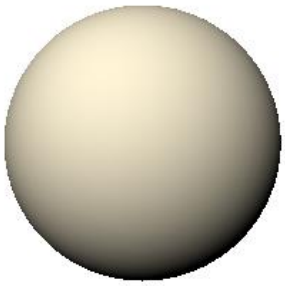
*Figura 5: Pinceles de Claybook*

El mundo se genera combinando los pinceles, cada uno de ellos tiene una traslación, rotación y escalado. Tiene una resolución de 1024x1024x512 de 8 bits con signo (586MB). Para simular el efecto arcilla, los objetos interactúan con el entorno y se mezclan mediante operadores booleanos de union y sustracción con suavizado exponencial mínimo/máximo. La combinación de los pinceles y sus operadores se estructura en un sistema de capas que permite ordenar cómo las operaciones se aplican, esto hace que se puedan modelar formas complejas a partir del conjunto de pinceles básicos.

*Figura 6: Claybook, 2018*

## 2.2 Funciones de distancia

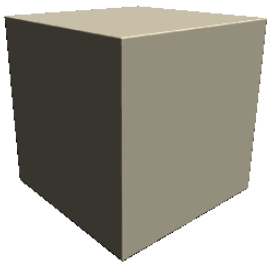
Las funciones de distancia son funciones $f: R^3 \rightarrow R$ que, dado un punto, permiten evaluar la distancia de este punto al volumen que representan. El signo de la función indica si el punto está dentro o fuera del volumen. Valores positivos se encuentran fuera del volumen, valores que tienden a cero se encuentran cerca de la superfície y valores negativos se encuentran dentro del volumen. Por ejemplo, la siguiente funcion de distancia con signo define una esfera con radio $r$ y $p(x, y, z)$ es el punto a evaluar:



```
// r: Radius.
float sdSphere( float3 p, float r )
{
  return length(p) - r;
}
```

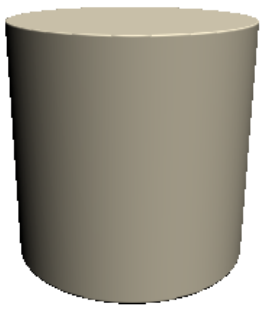
*Figura 7: Signed distance sphere*

El valor de distancia a la superfície de la esfera se calcula con la expresión $f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r$ . Esta función asume que la esfera está centrada en el origen. Por ejemplo, dado el punto $p = (1, 1, 0)$ y la esfera de radio $r = 1$ tenemos que la función se evalúa al valor $f(p) = 0.41$, por tanto, el punto $p$ se encuentra fuera del volumen de la esfera a una distancia de 0.41 unidades. En el artículo *Modeling with distance functions* [1], Íñigo Quílez [23] presenta una colección de funciones de distancia y operadores utilizados en muchas de sus demos hechas en Shadertoy. Algunas de las funciones de distancia propuestas son las siguientes:



```
// l: Length.
float sdBox( float3 p, float3 l )
{
  float3 d = abs(p) - l;
  return length(max(d, 0));
}
```

*Figura 8: Signed distance box*



```
// h: Height.
// r: Radius.
float sdCylinder( float3 p, float h, float r )
{
  float2 d = abs(float2(length(p.xz), p.y)) -
                 float2(r, h);
  return min(max(d.x, d.y), 0) + length(max(d, 0));
}
```

*Figura 9: Signed distance cylinder*

*Figura 10: Signed distance capsule*

```
// h: Height.
// r: Radius.
float sdCapsule( float3 p, float h, float r )
{
  p.y -= clamp(p.y, -h / 2, h / 2);
  return length(p) - r;
}
```

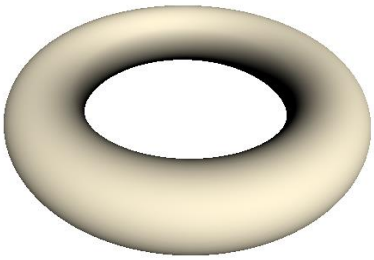


*Figura 11: Signed distance torus*

```
// d: Diameter.
// t: Thickness.
float sdTorus( float3 p, float d, float t )
{
  float2 q = float2(length(p.xz) - d, p.y);
  return length(q) - t;
}
```

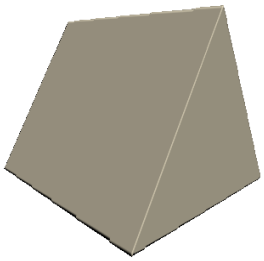


*Figura 12: Signed distance triangular prism*

```
// h: Height.
// w: Width.
float sdTriangularPrism( float3 p, float h,
                         float w )
{
  float3 q = abs(p);
  return max(q.z - w,
         max(q.x * 0.86 + p.y * 0.5, -p.y)
          - h * 0.5);
}
```

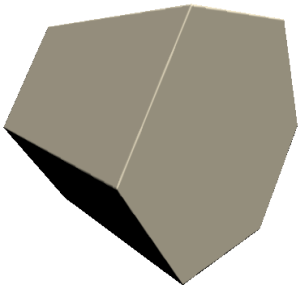


*Figura 13: Signed distance hexagonal prism*

```
// h: Height.
// w: Width.
float sdHexagonalPrism( float3 p, float h,
                        float w )
{
  const float3 k = float3(-0.86, 0.5, 0.57);

  p = abs(p);
  p.xy -= 2.0 * min(dot(k.xy, p.xy), 0.0) * k.xy;

  float2 d = float2(length(p.xy –
   float2(clamp(p.x, -k.z * h, k.z * h), h)) *
    sign(p.y - h), p.z - w);

  return min(max(d.x, d.y), 0) +
         length(max(d, 0));
}
```

Definir las primitivas mediante expresiones analíticas es una forma compacta de almacenar la información y permite tener precisión infinita en el rango del tipo de coma flotante elegido. Además, es posible aplicar otras funciones y operaciones que dan como resultado formas más complejas. Estas nuevas formas son generalmente más difíciles de modelar usando representaciones convencionales como las mallas de triángulos.

## 2.3 Operadores

Las primitivas definidas anteriormente están centradas en el origen de coordenadas en el espacio del objeto. Las operaciones básicas de traslación y rotación pueden realizarse sobre estas primitivas aplicando una matriz de transformación sobre el punto $p$ a evaluar, ya que ambas operaciones no distorsionan el espacio euclídeo. Por otra parte, escalar un objeto comprime o dilata el espacio euclídeo. Para escalar uniformemente la primitiva se divide el punto $p$ entre el factor de escala, se calcula la distancia a la primitiva y el resultado se multiplica de nuevo por el factor de escala. El escalado no uniforme provoca una distorsión en el espacio euclídeo, lo que provoca resultados erróneos en el cálculo de las distancias.

```
float scale( float3 p, float s )
{
  return SignedDistanceFunction(p / s) * s;
}
```

Otro conjunto de operadores que se pueden aplicar sobre las primitivas son los operadores booleanos, unión, intersección y resta. Estas operaciones no distorsionan el espacio y se implementan usando las funciones mínimo y máximo. Las siguientes figuras muestran el resultado de las operaciones de unión, intersección y resta entre una esfera y una caja:

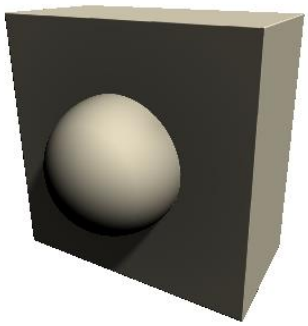


*Figura 14: Union operator*

```
float opUnion( float d1, float d2 )
{
  return min(d1, d2);
}
```

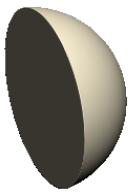


*Figura 15: Intersection operator*

```
float opIntersection( float d1, float d2 )
{
  return max(d1, d2);
}
```

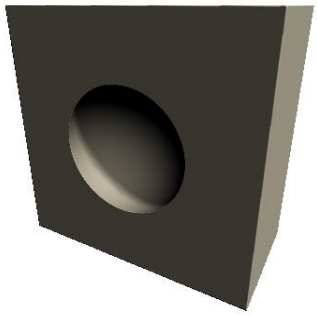


*Figura 16: Subtraction operator*

```
float opSubtraction( float d1, float d2 )
{
  return max(-d1, d2);
}
```

Estas operaciones son muy útiles para construir formas complejas, pero las formas resultantes tienen discontinuidades en sus derivadas, lo que provoca que el cambio de una superficie a otra sea drástico. Para el modelado de objetos reales es muy útil disponer de operadores que suavicen los bordes y controlen la interpolación de las primitivas.

El operador de suavizado mínimo [4] aproxima funciones resolviendo el problema de la discontinuidad de la función mínimo. Existen distintos métodos de suavizado con distintas características de suavizado y coste computacional. El suavizado polinómico y exponencial son los más comunes. Por ejemplo, dadas las funciones:

$$f(x) = 2\sin(2x)$$

$$g(x) = \frac{1}{\sqrt{x}} + 0.5$$

Representadas en las siguientes figuras:

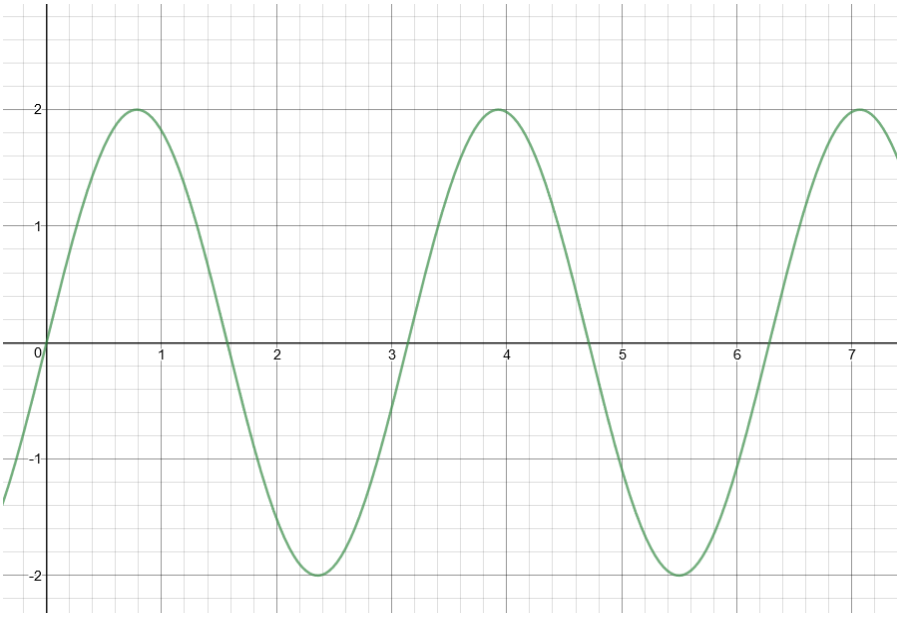



*Figura 17: Gráfica de la función f(x)*

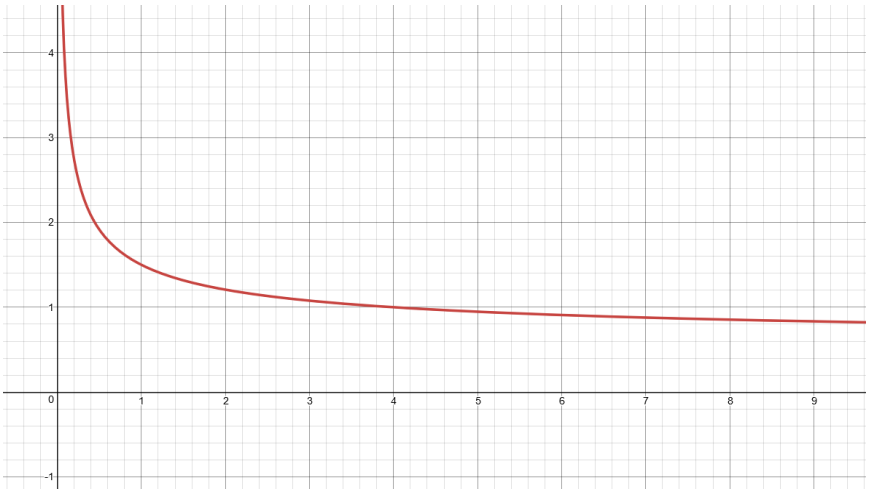



*Figura 18: Gráfica de la función g(x)*

Y el operador de suavizado mínimo polinómico:

$$h = clamp(0.5 + 0.5 * \frac{(b-a)}{k}, 0{,}1)$$

$$smin(a, b, k) = lerp(a, b, h) - k * h * (1 - h)$$

Donde $a$ y $b$ son los valores a interpolar, $k$ el factor que controla el suavizado, $lerp$ es la función de interpolación lineal y $clamp$ es una función que restringe el valor resultado al rango [0, 1].

Con el valor de suavizado $k = 1$ se obtiene la función interpolada (en negro):

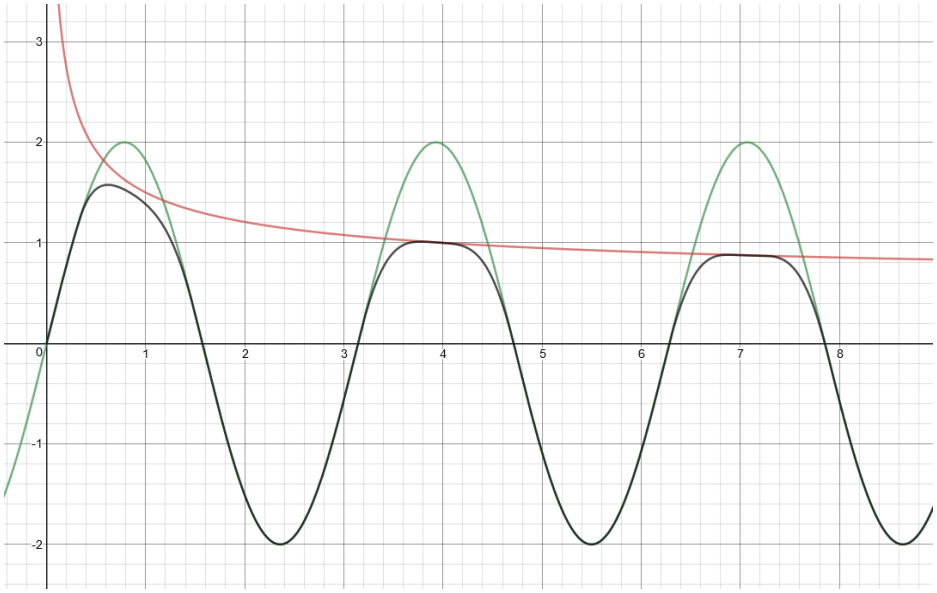



*Figura 19: Operador de suavizado mínimo con k=1*

Si se aplica el operador con el valor de suavizado $k = 0$ se obtiene como resultado la función mínimo, sin suavizado (en negro).

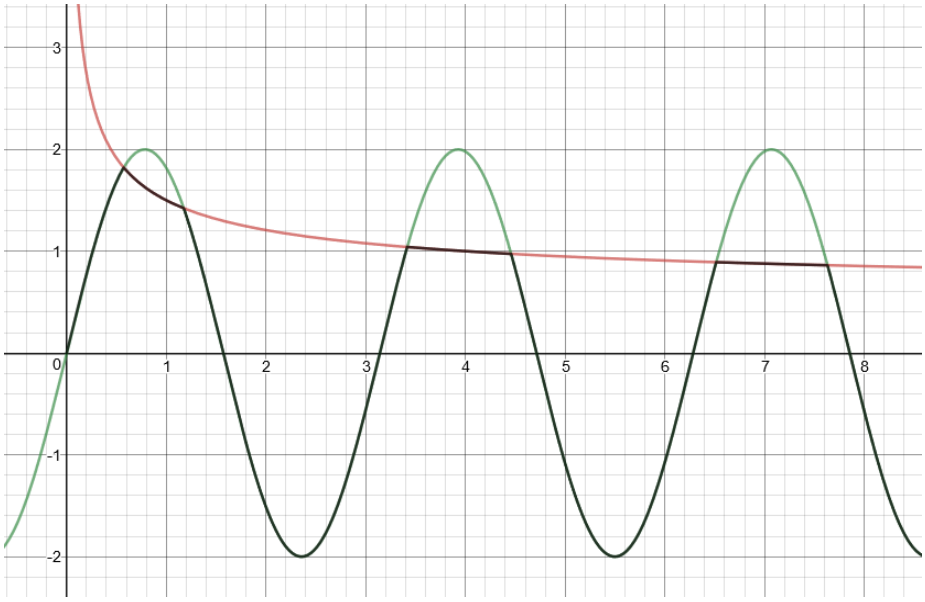



*Figura 20: Operador de suavizado mínimo con k=0*

Los siguientes operadores implementan el suavizado mínimo/máximo polinómico y tienen el parámetro $k$ para controlar el suavizado. Estos operadores no son conmutativos (dependiendo del orden de las distancias el resultado es distinto) y tienen discontinuidades de segundo orden.

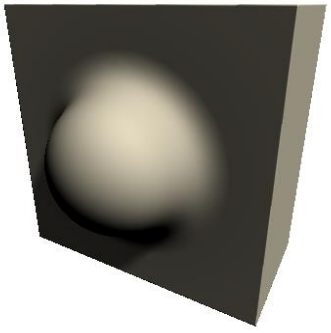


*Figura 21: Smooth union operator with k = 0.5*

```
float opSmoothUnion( float d1, float d2, float k )
{
  float h = clamp(0.5 + 0.5 * (b - a) / k, 0, 1);
  return lerp(b, a, h) - k * h * (1 - h);
}
```

*Figura 22: Smooth intersection operator with k = 0.5*

```
float opSmoothIntersection( float d1, float d2, float k )
{
  float h = clamp(0.5 - 0.5 * (b - a) / k, 0, 1);
  return lerp(b, a, h) + k * h * (1 - h);
}
```

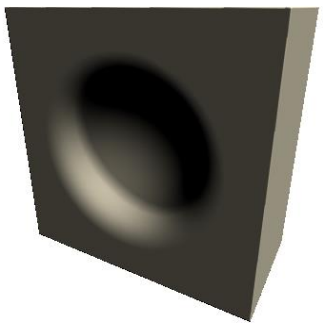


*Figura 23: Smooth subtraction operator with k = 0.5*

```
float opSmoothSubtraction( float d1, float d2, float k )
{
  d1 = -d1;
  float h = clamp(0.5 - 0.5 * (b - a) / k, 0, 1);
  return lerp(b, a, h) + k * h * (1 - h);
}
```

Como se puede observar, los resultados son más orgánicos y naturales que las formas obtenidas usando los operadores booleanos sin suavizado.

Otro conjunto de operaciones que se pueden realizar sobre las funciones de distancia son los desplazamientos y deformaciones. Estos operadores permiten obtener formas complejas, pero deforman el espacio euclídeo convirtiéndolo en un espacio curvo. Para obtener resultados correctos en el algoritmo de raymarching es necesario rectificar el parámetro de tamaño de paso, decrementándolo en un valor inversamente proporcional al factor con el que se distorsiona el espacio.

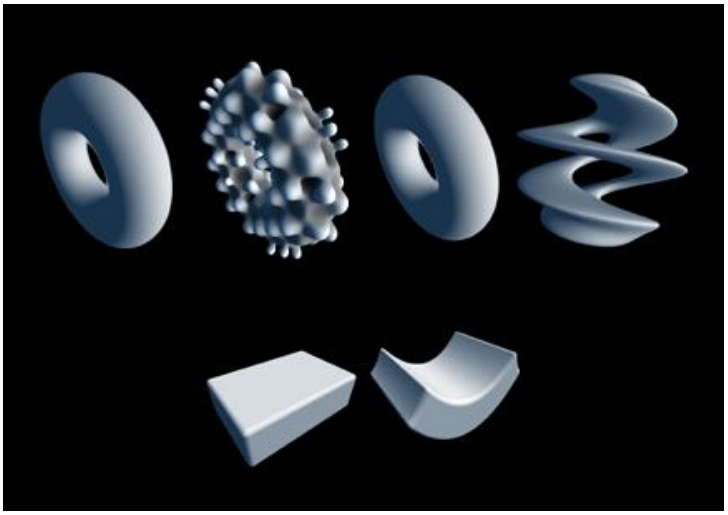


*Figura 24: Operadores de desplazamiento, giro y curva*

Las operaciones de desplazamiento usan funciones senoidales como patrón para desplazar los puntos de la superficie de las primitivas y suelen usarse en función del tiempo para animar los objetos. Los operadores de giro rotan los puntos en un eje. En el ejemplo anterior el toro se ha girado en el eje Y. Los operadores de curva son similares a los de giro, pero aplicando la rotación sobre distintos ejes.

También pueden aplicarse a las funciones de distancia otras operaciones como la simetría o la repetición. Estos operadores tienen la ventaja de que el cálculo de la función de distancia sólo se realiza una vez y por tanto pueden hacerse infinitas repeticiones. En las siguientes imágenes pueden verse la aplicación de ambos operadores.

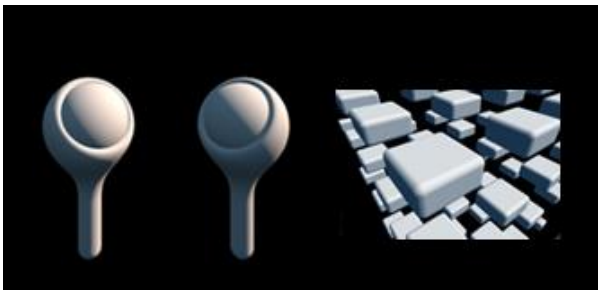


*Figura 25: Operadores de simetría y repetición*

## 2.4 Raymarching

Raymarching [2] es una técnica iterativa que calcula la intersección de un rayo con una superficie dando pasos en la dirección del rayo. Se utiliza cuando la superficie objetivo no es sencilla de calcular. Mientras que en el trazado de rayos se calcula la intersección del rayo con la superficie, en raymarching se marcha a pasos en la dirección del rayo hasta que se intersecta con la superficie. Las funciones de distancia se renderizan usando esta familia de algoritmos.

El algoritmo de raymarching básico avanza el rayo con un tamaño de paso fijo y en cada uno evalúa las funciones de distancia para saber si el punto intersecta con la superficie. Si el valor de distancia es menor que un valor pequeño $epsilon$ , entonces se toma ese punto como intersección con la superficie.

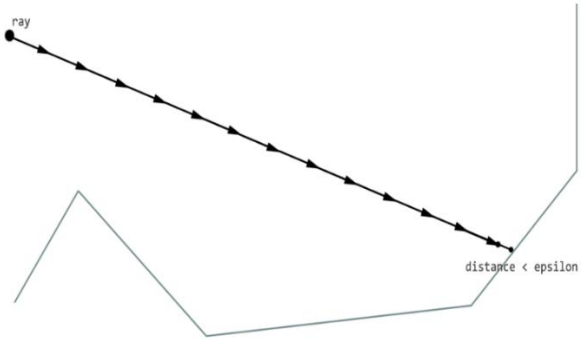



*Figura 26: Raymarching con tamaño de paso fijo*

Este proceso es computacionalmente costoso, ya que se realiza la marcha del rayo por cada píxel del buffer de dibujado destino. Además, es necesario controlar los casos donde el rayo nunca intersecta, con un parámetro de pasos máximos o distancia máxima. La calidad y eficiencia del algoritmo depende en gran medida de un parámetro de mínima distancia. Si el valor es grande se darán menos pasos y será más rápido, pero es posible que las intersecciones no se calculen correctamente y la calidad del resultado final sea peor.

```
Raymarch( float3 origin, float3 direction )
{
	float t = 0;
	for( int i = 0; i < STEPS; i++ )
	{
		if( t > MAX_DISTANCE )
		{
			break;
		}

		float3 p = origin + direction * t;
		float d = SignedDistanceField(p);
		if( d < MIN_DISTANCE )
		{
			return Shade(p);
		}

		t += STEP_SIZE; // Fixed step size.
	}

	return Color.white;
}
```

## 2.5 Sphere tracing

Una mejora del algoritmo básico de raymarching es el trazado de esferas o sphere tracing [11]. El algoritmo consiste en evaluar en cada paso las funciones de distancia y usar este valor de distancia como tamaño del siguiente paso. También se le suele llamar raymarching asistido por distancia. Este valor de distancia garantiza que el rayo no intersecta ningún objeto, ya que la intersección más cercana se encuentra a la distancia calculada. Esto hace que el número de pasos disminuya significativamente, mejorando el rendimiento y evitando errores de precisión cuando la marcha del rayo se encuentra cerca de los objetos del campo de distancia.

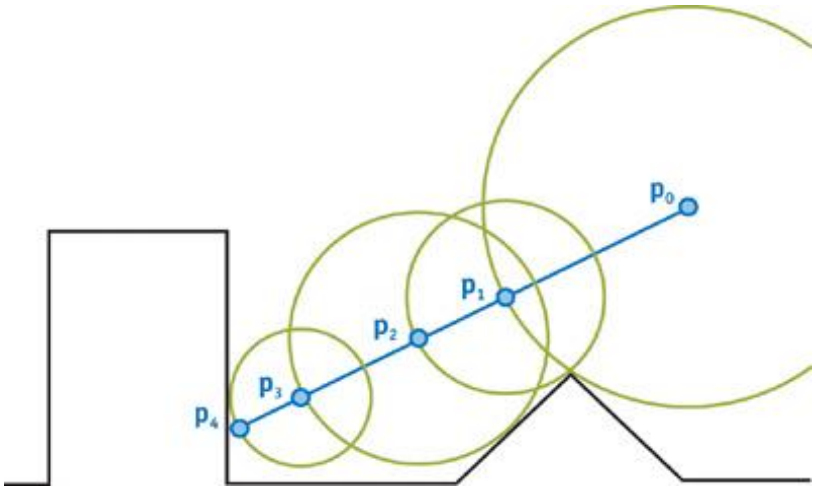



*Figura 27: Sphere tracing, GPU Gems 2*

El pseudocódigo del algoritmo de raymarching con la mejora de la marcha asistida por distancia sería:

```
Raymarch( float3 origin, float3 direction )
{
	float t = 0;
	for( int i = 0; i < STEPS; i++ )
	{
		if( t > MAX_DISTANCE )
		{
			break;
		}

		float3 p = origin + direction * t;
		float d = SignedDistanceField(p);
		if( d < MIN_DISTANCE )
		{
			return Shade(p);
		}

		t += d; // Distance-aided.
	}

	return Color.white;
}
```

## 2.6 Iluminación y sombreado

Las técnicas y modelos de iluminación como la reflectancia de Lambert o Blinn-Phong también se usan para el renderizado de las funciones de distancia. Estas técnicas calculan el color de cada píxel en función de una serie de cálculos realizados sobre vectores. Estos vectores típicamente son:

- V: Vector de la vista.
- L: Vector de la fuente de luz.
- N: Vector normal a la superficie en un punto.

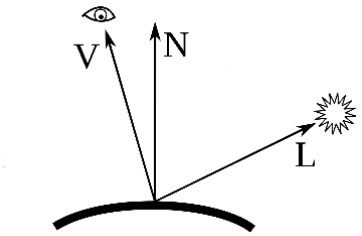



*Figura 28: Vectores V, L y N*

Al contrario que las mallas de triángulos, las funciones de distancia no tienen información sobre la normal a la superficie en el punto a evaluar. El cálculo del vector N se realiza evaluando el gradiente en puntos cercanos al punto objetivo. En el artículo *Numerical normals for SDFs* [3], Íñigo Quílez propone distintos métodos para el cálculo de las normales. En la implementación de los métodos se asume que la función $f(p)$ devuelve la distancia del punto $p$ al conjunto de funciones de distancia, también llamado campo de funciones de distancia.

El método de diferencias centradas evalúa las funciones de distancia en 6 puntos cercanos al punto objetivo. Estos puntos están desplazados, y por tanto centrados, del punto objetivo un valor $h$ muy pequeño en los sentidos positivo y negativo de los ejes.

```
float3 CalculateNormalCentralDifferences( float3 p, float h )
{
   float3 N = normalize(float3(
      f(p + float3(h, 0, 0)) - f(p - float3(h, 0, 0)),
      f(p + float3(0, h, 0)) - f(p - float3(0, h, 0)),
      f(p + float3(0, 0, h)) - f(p - float3(0, 0, h))
   ));

   return N;
}
```

En algunos casos, la evaluación 7 veces de las funciones de distancia es muy costosa y se usa el método de diferencias hacia adelante. Este método evalúa las funciones de distancia en el punto objetivo y en los puntos desplazados un valor $h$ en los sentidos positivos de los ejes, dando lugar a 4 evaluaciones.

```
float3 CalculateNormalForwardDifferences( float3 p, float h )
{
   float d = f(p);
   float3 N = normalize(float3(
      f(p + float3(h, 0, 0)) - d,
      f(p + float3(0, h, 0)) - d,
      f(p + float3(0, 0, h)) - d
   ));

   return N;
}
```

Una alternativa propuesta por Paul Malin [24] en Shadertoy es usar un tetrahedro como puntos a evaluar. Este método, al igual que con las diferencias hacia adelante, evalúa las funciones de distancia 4 veces.

```
float3 CalculateNormalTetrahedron( float3 p, float h )
{
   const float2 k = float2(1, -1);
   float3 N = normalize(float3(
      k.xyy * f(p + k.xyy * h) +
      k.yyx * f(p + k.yyx * h) +
      k.yxy * f(p + k.yxy * h) +
      k.xxx * f(p + k.xxx * h)
   ));

   return N;
}
```

Una vez calculado el vector N ya es posible aplicar cualquier técnica de iluminación que se base en operaciones sobre los vectores V, L y N, como el cálculo de componentes difusas o especulares de un objeto siguiendo los modelos de reflectancia de Lambert o Blinn-Phong.

Por otra parte, técnicas modernas de sombreado realista como las sombras suaves [6] o la oclusión ambiental son fáciles de implementar en un raymarcher de funciones de distancia. El cálculo de las sombras se realiza marchando sobre el vector L (vector de la fuente de luz) hasta, como máximo, el punto objetivo donde se va a sombrear o hasta que se encuentra una intersección, devolviendo el valor 0 o 1 y produciendo sombras fuertes.

```
float HardShadows( float3 origin, float3 direction )
{
   for( float t = MIN; t < MAX; )
   {
      float h = f(origin + direction * t);
      if( h < MIN_DISTANCE )
      {
         return 0;
      }

      t += h;
   }

   return 1;
}
```

Para crear sombras suaves también se marcha sobre el vector L y se comprueba la distancia a la que se encontraba la intersección con respecto al punto objetivo, haciendo que cuanto más cercana se encuentre la intersección al punto, más oscura sea la sombra. En cada paso se calcula un valor de penumbra y se comprueba si ese valor es más pequeño que los anteriores, y al final se devuelve el valor mínimo. Además, se puede añadir un parámetro (en el código propuesto llamado PENUMBRA) para controlar el rango de penumbra de la sombra, es decir, cómo de suave es la sombra. Valores altos hacen que el método devuelva resultados similares a los producidos con sombras fuertes.

```
float SoftShadows( float3 origin, float3 direction )
{
   float result = 1;
   for( float t = MIN; t < MAX; )
   {
      float h = f(origin + direction * t);
      if( h < MIN_DISTANCE )
      {
         return 0;
      }

      result = min(result, PENUMBRA * h / t);
      t += h;
   }

   return result;
}
```

Por otra parte, el cálculo de la oclusión ambiental se realiza marchando sobre el vector N, normal a la superficie, desde el punto objetivo un número de iteraciones. Si durante la marcha se intersecta con el campo de distancia, el punto está bloqueado y por tanto se añade un valor de oclusión en función de la distancia atravesada, haciendo que valores cercanos sean más oscuros y valores lejanos sean más claros.

```
float AmbientOcclusion( float3 position, float3 normal )
{
   float ao = 0;

   for( int i = 1; i < ITERATIONS; i++ )
   {
      float d = STEP_SIZE * i;
      ao += max(0, (d - f(position + normal * d)) / d);
   }

   return 1 - ao * INTENSITY;
}
```

El cálculo de reflexiones sobre un punto que ha intersectado con el campo de funciones de distancia se realiza haciendo una nueva marcha sobre un nuevo rayo reflejado resultado de reflejar el rayo incidente y la normal en el punto intersectado. Realizar una nueva marcha incrementa el coste computacional del algoritmo y hace que escenas más complejas no se puedan visualizar en tiempo real.

# 3. Diseño e implementación

La implementación del raymarcher para renderizar campos de distancia se ha realizado sobre el motor de videojuegos Unity, versiones 2018 y 2019. El motor cuenta con un potente compilador de shaders que permite usar la mayoría de características que ofrecen las GPUs modernas. Los shaders se escriben en el lenguaje HLSL encapsulado por el lenguaje declarativo ShaderLab (ficheros shader). ShaderLab contiene metadatos que permiten configurar parámetros como el tipo de blending, el buffer de profundidad o los pases que ejecuta el shader.

Los shaders se compilan de HLSL a los distintos lenguajes de las plataformas objetivo. Para plataformas que usan HLSL el código se compila directamente a DirectX (D3D11, D3D12, etc.) y para el resto de plataformas se utiliza el compilador cruzado HLSLcc. Este compilador obtiene de entrada código bytecode DirectX que se traduce a los lenguajes GLSL, GLSL ES, GLSL para Vulkan o Metal Shading Language.

## 3.1 Estructura y componentes

El sistema de renderizado de funciones de distancia mediante raymarching se ha implementado como un paquete de Unity con la siguiente jerarquía de directorios:

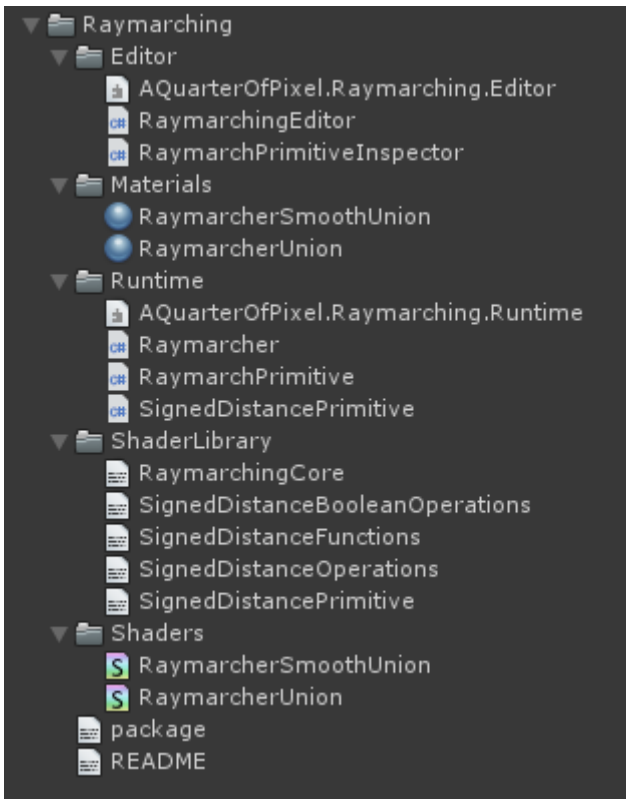



*Figura 29: Jerarquía de directorios*

Los directorios **Editor** y **Runtime** contienen el código C# que se ejecuta en la CPU, ambos con sus respectivas definiciones de ensamblados (*assembly definitions*). Las deficiones de ensamblados son un sistema de definición de librerías dinámicas (*dynamic link libraries* o *dll*). La utilización de librerías dinámicas permite estructurar y separar el código que se compila en el editor del que se compila en runtime, evita conflictos de nombres y hace el código más modular y mantenible. En la figura anterior pueden verse las definiciones de ensamblados **Raymarching.Editor** y **Raymarching.Runtime**.

Los directorios **ShaderLibrary** y **Shaders** contienen el código HLSL que se ejecuta en la GPU, separado en core y shaders de raymarching básicos respectivamente. Por último, el directorio **Materials** contiene materiales que usan los shaders de raymarching básicos.

La creación de primitivas de las funciones de distancia se hace mediante el uso del componente **RaymarchPrimitive** en un gameobject. Este componente tiene como parámetros el tipo de primitiva, sus dimensiones y el material con el que se va a renderizar la primitiva.

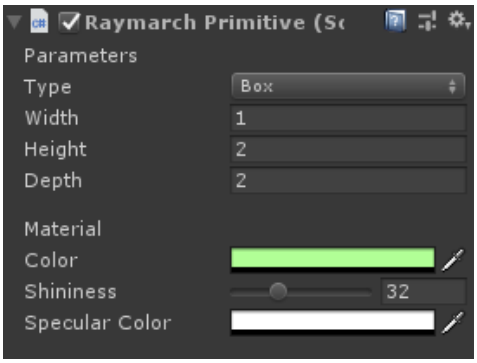



*Figura 30: Raymarch primitive inspector*

Las primitivas disponibles son:

- Plano.
- Caja.
- Esfera.
- Elipse.
- Cilindro.
- Cápsula.
- Toro.
- Prisma triangular.
- Prisma hexagonal.

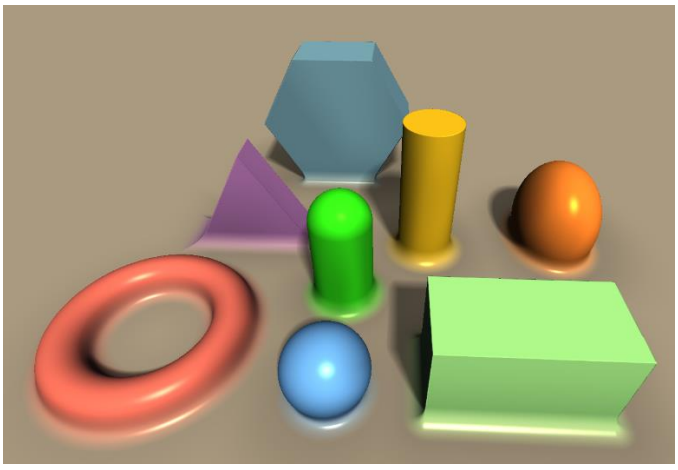


*Figura 31: Conjunto de primitivas*

El material contiene los parámetros del modelo de iluminación de **Blinn-Phong**: el color de la primitiva, el color de los reflejos especulares y factor de brillo de estos.

La lista de primitivas de la escena se añade al componente **Raymarcher** que se encuentra en la cámara que renderiza la escena. Este componente se encarga de enviar a la GPU los parámetros de los algoritmos de raymarching, iluminación, sombreado y oclusión ambiental.

El algoritmo de raymarching tiene como parámetros el material del shader que se utiliza para realizar la marcha, los pasos máximos que se hacen sobre el rayo, la distancia mínima para saber si se ha intersectado la primitiva y la distancia máxima que se avanza en el rayo. Además, existe un parámetro para colorear el buffer de color destino con el número de pasos que se han dado en el algoritmo.

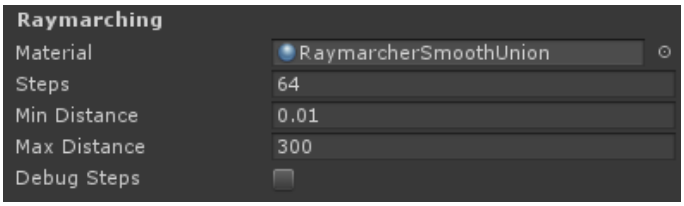



*Figura 32: Parámetros de raymarching*

Los parámetros de iluminación son un desplazamiento para el cálculo de las normales, el algoritmo del cálculo de las normales, si existe o no luz ambiental y si esta es proporcionada por el motor de Unity o es un color ambiental propio y si se calcula la iluminación especular.

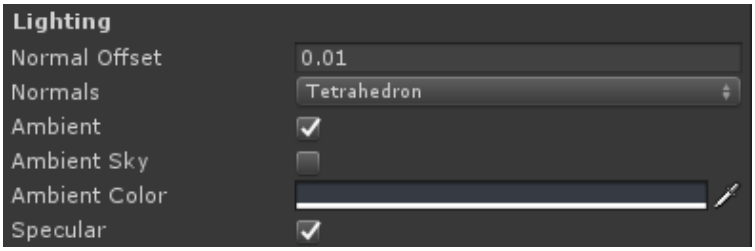



*Figura 33: Parámetros de iluminación*

Los parámetros del cálculo de sombras son el algoritmo usado, el intervalo de distancia donde se calculan las sombras, la intensidad de la sombra y el factor de penumbra.


*Figura 34: Parámetros de las sombras*

Los parámetros del cálculo de la oclusión ambiental son el tamaño de paso del algoritmo, el número de iteraciones y la intensidad de la sombra producida por la oclusión.


*Figura 35: Parámetros de la oclusión ambiental*

El componente está adjunto a una cámara, ya que el algoritmo de raymarching necesita la información de la cámara para realizar la marcha de los rayos. Los datos necesarios de la cámara son su posición en el espacio de coordenadas del mundo, la matriz de transformación del espacio de coordenadas de la vista a la del mundo, el frustum de la cámara y los buffers de profundidad y de color de la escena de Unity.

La marcha se realiza sobre un quad posicionado delante de la cámara. Los parámetros necesarios son la posición de la cámara en el espacio del mundo, la matriz de transformación de la cámara de la vista al mundo y el frustum de la cámara. Por último, se envía el buffer con las primitivas a renderizar.

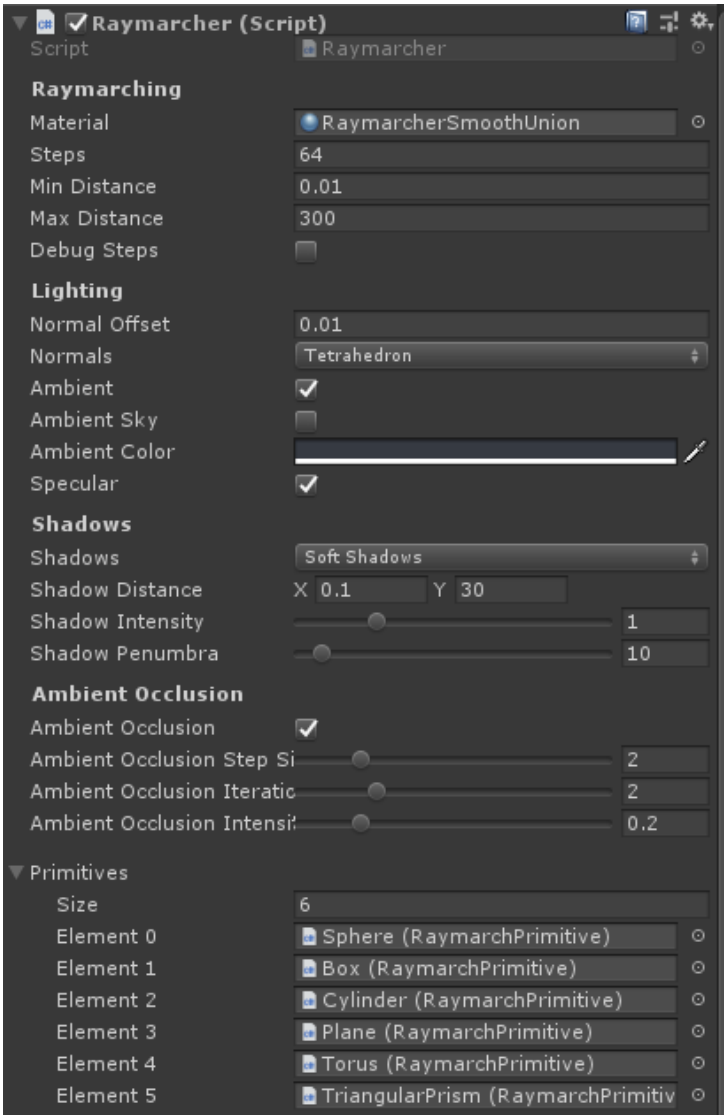



*Figura 36: Raymarcher inspector*

## 3.2 Raymarching en la GPU

A la GPU se le envían los datos de los algoritmos de raymarching, iluminación, sombreado, oclusión ambiental y el buffer que contiene la lista de estructuras **SignedDistancePrimitive**. Esta estructura está compuesta por:

- Tipo de primitiva.
- Matriz de transformación del espacio del mundo al objeto.
- Escala de la primitiva.
- Parámetros que la definen.
- Parámetros del material con la que se renderiza.

```
public struct SignedDistancePrimitive
{
    public enum Type
    {
        Plane = 0,
        Box = 1,
        Sphere = 2,
        Ellipsoid = 3,
        Cylinder = 4,
        Capsule = 5,
        Torus = 6,
        TriangularPrism = 7,
        HexagonalPrism = 8
    }

    public Type type;
    public Matrix4x4 transform;
    public float scale;
    public float parameter0;
    public float parameter1;
    public float parameter2;
    public Color color;
    public float shininess;
    public Color specularColor;
}
```

La estructura anterior tiene su equivalente en la GPU. La estructura también se llama **SignedDistancePrimitive** y tiene las mismas variables que su equivalente en la CPU.

```
struct SignedDistancePrimitive
{
       int type;
       float4x4 transform;
       float scale;
       float parameter0;
       float parameter1;
       float parameter2;
       float4 color;
       float shininess;
       float4 specularColor;
};
```

El fichero **RaymarchingCore** contiene el código HLSL del raymarcher, esto es, la declaración de los datos que se envían a la GPU, los shaders de vértice y fragmento, y los algoritmos y estructuras de datos usados para implementar tanto el raymarcher como el renderizado. La implementación de las funciones de distancia y los operadores se encuentran en los ficheros **SignedDistanceFunctions**, **SignedDistanceOperations** y **SignedDistanceBooleanOperations** (ver anexo). Estos ficheros están incluidos en el fichero **RaymarchingCore** y están implementados como una librería que se puede usar por otros sistemas.

Dado que los algoritmos de raymarching y renderizado tienen muchos parámetros, se ha usado la característica de variantes de shaders de Unity. Este sistema permite mantener parte del código de shaders fijo y producir variantes con modificaciones realizadas a partir de directivas de preprocesado que se activan y desactivan en la CPU. Las directivas usadas son las siguientes:

```
#pragma multi_compile _ DEBUG_STEPS
#pragma multi_compile _ NORMALS_FORWARD_DIFFERENCES NORMALS_TETRAHEDRON
#pragma multi_compile _ AMBIENT
#pragma multi_compile _ SPECULAR
#pragma multi_compile _ AMBIENT_OCCLUSION
#pragma multi_compile _ HARD_SHADOWS SOFT_SHADOWS SOFT_SHADOWS_IMPROVED
```

La directiva **DEBUG_STEPS** permite visualizar los pasos que ha dado el raymarcher. Las directivas de tipo **NORMALS** permite seleccionar el algoritmo de cálculo de las normales. Las directivas **AMBIENT** y **SPECULAR** activan o desactivan la luz ambiental y especular. La directiva **AMBIENT_OCCLUSION** activa o desactiva la oclusión ambiental. Las directivas **SHADOWS** controlan el tipo de algoritmo usado para calcular las sombras.

El algoritmo de raymarching se realiza sobre un quad posicionado delante de la cámara. Por tanto, el shader de vértice se ejecuta 4 veces, una por cada vértice. Los cálculos realizados en el shader de vértice son transformar el vértice del espacio del objeto al de clip y calcular la dirección del rayo a partir del frustum de la cámara. La marcha de los rayos se realiza en el shader de fragmento llamando a la función **Raymarch** que tiene como entrada el origen y dirección del rayo sobre el que se va a hacer la marcha, la coordenada de textura del quad y el valor de profundidad de la escena de Unity en ese píxel.

```
fixed4 Raymarch( float3 origin, float3 direction, float2 uv, float depth )
```

La ejecución de todos los shaders de fragmento da como salida un buffer de color con las primitivas renderizadas en la escena. Esta escena se compara con el buffer de profundidad para evitar renderizar primitivas que se encuentran detrás de objetos renderizados por el motor de Unity. El bucle principal de la marcha realiza un máximo de **_Steps** iteraciones o pasos.

```
for( i = 0; i < _Steps; i++ )
```

En cada paso se comprueba si la distancia marchada supera el valor de profundidad **depth**. Si esto es así se para la marcha ya que existe un objeto nativo de Unity delante del rayo. También se comprueba si la marcha ha superado una distancia máxima definida como **_MaxDistance**. Si todas estas comprobaciones tienen éxito se realiza la marcha sobre el rayo.

```
float3 p = origin + direction * t;
```

Una vez se ha realizado la marcha sobre el rayo, se calcula la distancia al campo de distancias.

```
RaymarchData data = SignedDistanceField(p);
```

La función **SignedDistanceField** se implementa en los shaders. La entrada es el punto a evaluar el campo de distancias y la salida es un struct **RaymarchData** que contiene la información de la distancia a la que se encuentra el objeto más cercano al campo de distancias y los valores que se usarán para renderizar el objeto.

Los shaders que describen un campo de distancias deben implementar la función **SignedDistanceField**. Un ejemplo básico es el shader **RaymarcherShader** que itera sobre el conjunto de primitivas, transforma la posición del espacio del mundo al local de la primitiva y calcula la función de distancia de cada una (ver anexo con la lista de funciones de distancia). Por cada primitiva se comprueba si su valor de distancia es menor que la distancia anterior, si es así entonces actualiza los valores de distancia mínima y los parámetros de renderizado. El resultado de calcular el mínimo de todas las distancias es equivalente a realizar una unión sobre todas las primitivas.

```
RaymarchData SignedDistanceField( float3 position )
{
	RaymarchData data;
	float mind = 0;

	for( int i = 0; i < _PrimitivesBufferSize; i++ )
	{
		SignedDistancePrimitive primitive = _PrimitivesBuffer[i];

		// Transform the position to object space.
		float3 p = mul(primitive.transform, float4(position, 1)).xyz;

		// Calculate the distance.
		float d = SignedDistanceFunction(primitive, p);

		if( i == 0 )
		{
			data.distance = d;

			data.color = primitive.color;
			data.shininess = primitive.shininess;
			data.specularColor = primitive.specularColor;

			mind = data.distance;
		}
		else
		{
			data.distance = opUnion(data.distance, d);

			if( d < mind )
			{
				data.color = primitive.color;
				data.shininess = primitive.shininess;
				data.specularColor = primitive.specularColor;

				mind = d;
			}
		}
	}

	return data;
}
```

Una vez se devuelve el resultado de la evaluación del campo de distancias al bucle principal del raymarcher se comprueba si la distancia devuelta es menor que un umbral **_MinDistance**, usado para evitar errores numéricos de redondeo. Si es menor entonces se calcula la normal en ese punto con uno de los métodos propuestos:

```
#if defined(NORMALS_FORWARD_DIFFERENCES)
{
	N = CalculateNormalForwardDifferences(p, _NormalOffset);
}
#elif defined(NORMALS_TETRAHEDRON)
{
	N = CalculateNormalTetrahedron(p, _NormalOffset);
}
#else
{
	N = CalculateNormalCentralDifferences(p, _NormalOffset);
}
#endif
```

Con la normal calculada se llama a la función **Shade** que renderiza el píxel dados los datos de renderizado del campo de distancias.

```
fixed4 Shade( float3 position, float3 normal, RaymarchData data )
```

Una vez se ha calculado el color del punto donde se ha intersectado con el campo de distancias se usa como valor de retorno del shader de fragmento. Por el contrario, si no se ha intersectado con el campo de distancias, se devuelve el valor de color del frame buffer calculado por la tubería de renderizado de Unity.

## 3.3 Iluminación

La iluminación está implementada siguiendo el modelo de reflexión de **Blinn-Phong** para iluminar el píxel, esto es, se calcula el producto escalar del vector normal con el vector de la luz. Este valor multiplicado con el color de la luz da la componente difusa de la luz. La componente ambiental es opcional. Si se activa se suma a la componente difusa. Para el cálculo de la componente especular se hace el producto escalar del vector de la vista con el de reflexión (vector de la luz reflejado con la normal). Este valor se eleva a un parámetro de brillo para controlar la intensidad del brillo y finalmente se multiplica por el color de la luz difusa y el color de la luz especular.

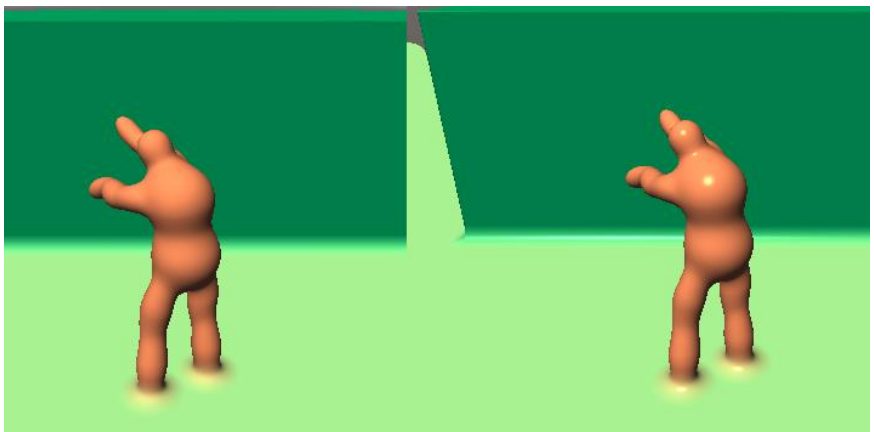


*Figura 37: Iluminación difusa y especular*

## 3.4 Sombras

Para el cálculo de las sombras se han implementado 3 métodos, sombras duras, sombras suaves y sombras suaves mejoradas. El cálculo de las sombras se realiza haciendo raymarching desde el punto objetivo que se va a iluminar sobre el vector de la luz hasta que se intersecta con un objeto del campo de distancias. Este proceso da como resultado un valor de sombreado entre 0 y 1. Si se quieren sombras duras este valor es entero y si se quieren suaves el valor es un número real. El sombreado se controla mediante un parámetro de intensidad que se utiliza elevando el resultado del cálculo de la sombra a este parámetro. En la siguiente figura puede verse la diferencia de los algoritmos de sombras duras y suaves:

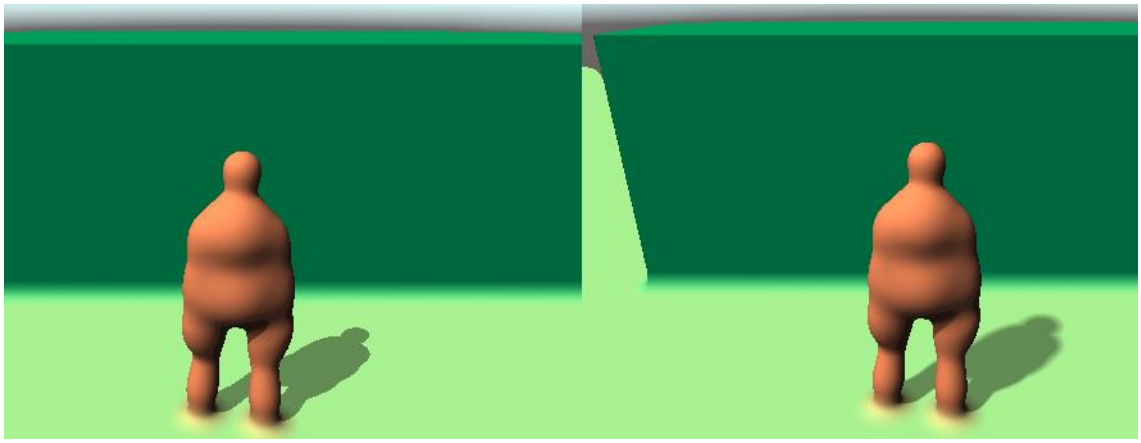


*Figura 38: Sombras duras y suaves*

## 3.5 Oclusión ambiental

El cálculo de la oclusión ambiental se realiza marchando desde el punto objetivo sobre el vector normal a la superfície un número de iteraciones. Si durante esa marcha se intersecta de nuevo el campo de distancias, eso significa que el punto está ocluído por otro objeto. Este valor se controla por un parámetro que indica la intensidad de la oclusión ambiental.

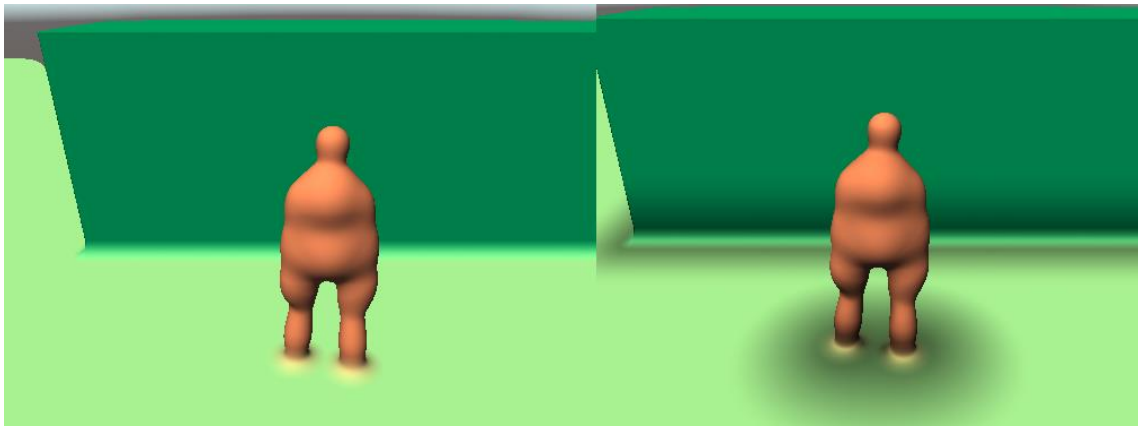


*Figura 39: Oclusión ambiental*

# 4. Demostraciones y tests

Para demostrar el uso del paquete de raymarching en Unity se ha creado un proyecto con un conjunto de escenas donde se hace uso del mismo. En todas ellas se utiliza el plugin de Unity **Graphy**, que muestra información sobre el hardware donde se ejecuta el proyecto, el rendimiento en frames por segundo, memoria utilizada, asignada y reservada y la resolución a la que se está renderizando. En los siguientes apartados se describen las escenas utilizadas.

## 4.1 Primitives

En la escena **Primitives** se pueden ver algunas de las primitivas definidas como funciones de distancia; un toro, un prisma triangular, una caja, una esfera y un cilindro respectivamente. Estas primitivas se mezclan usando una unión suavizada. En la siguiente imagen se puede ver el resultado final:

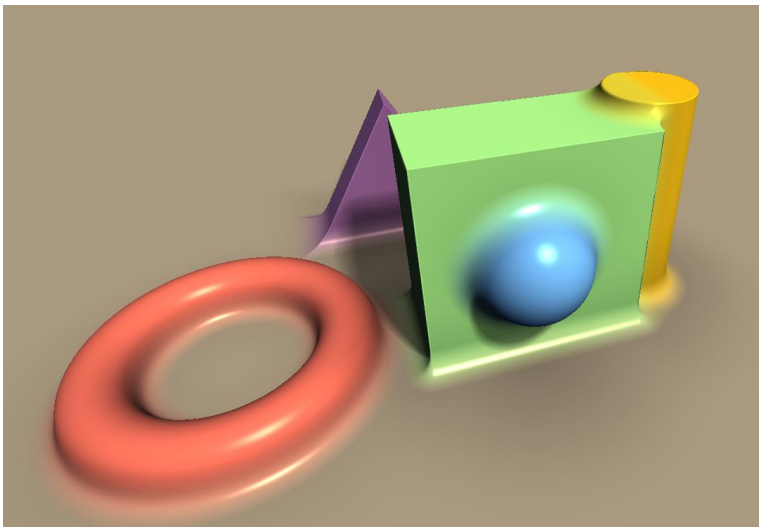


*Figura 40: Escena con primitivas mezcladas con una unión suavizada*

## 4.2 MIDI

En las escenas **MIDI** se renderizan un conjunto de primitivas que se escalan, desplazan y deforman de forma interactiva mediante un controlador MIDI. MIDI (*Musical Instrument Digital Interface*) es un estándar tecnológico que permite que instrumentos electrónicos y ordenadores se comuniquen entre sí. El controlador o instrumento electrónico MIDI usado es un Akai Lpd8 y meidante el plugin de Unity **MidiJack** desarrollado por Keijiro Takahashi de Unity Technologies Japón se pueden leer los valores de entrada.


*Figura 41: Controlador Midi Akai Lpd8*

Los raymarcher shaders usados en estas escenas exponen parámetros que deforman y desplazan las primitivas donde, por ejemplo, dado el punto de intersección con el campo de distancias, se desplaza la posición usando la siguiente función senoidal.

```
float opDisplace( float3 p )
{
	float ax = 2;
	float ay = 2;
	float az = 2;
	float fx = _SinTime.w;
	float fy = _SinTime.w;
	float fz = _SinTime.w;
	float d = ax * sin(fx * p.x) * ay * sin(fy * p.y) * az * sin(fz * p.z);
	return d;
}
```

Las siguientes imágenes muestra una secuencia de fotogramas de las escenas donde se escalan y deforman las primitivas. Las primitivas se renderizan con el operador de unión suavizada.

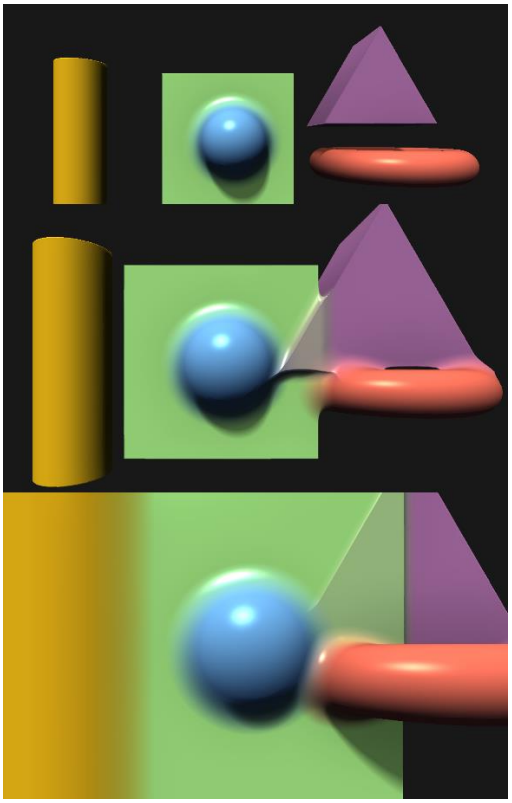


*Figura 42: MIDI scale scene*

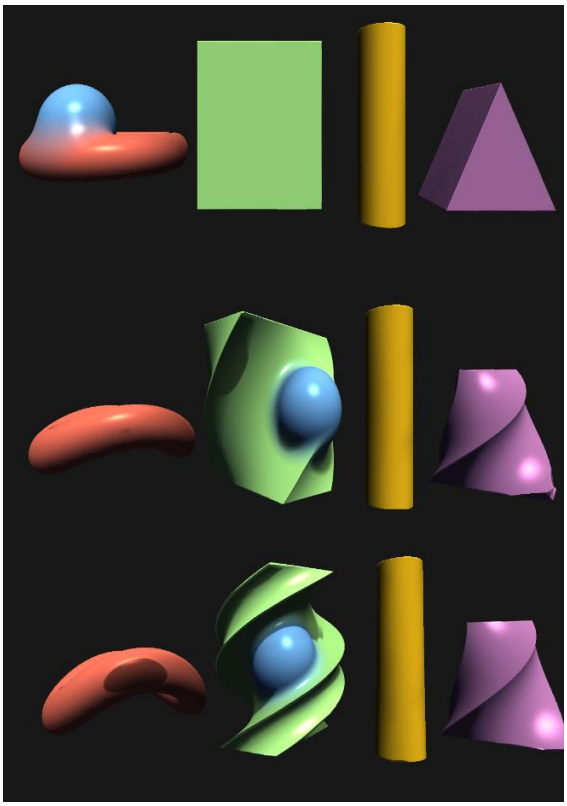


*Figura 43: MIDI twist scene*

## 4.3 Character

La escena **Character** demuestra cómo es posible usar el sistema de raymarching con sistemas nativos de Unity como Mecanim para animar personajes. El personaje base está preparado para realizar animaciones de humanoide y se le han creado primitivas de raymarching en los huesos, mezcladas usando una unión suavizada. El resultado final se puede ver en las siguientes imágenes:

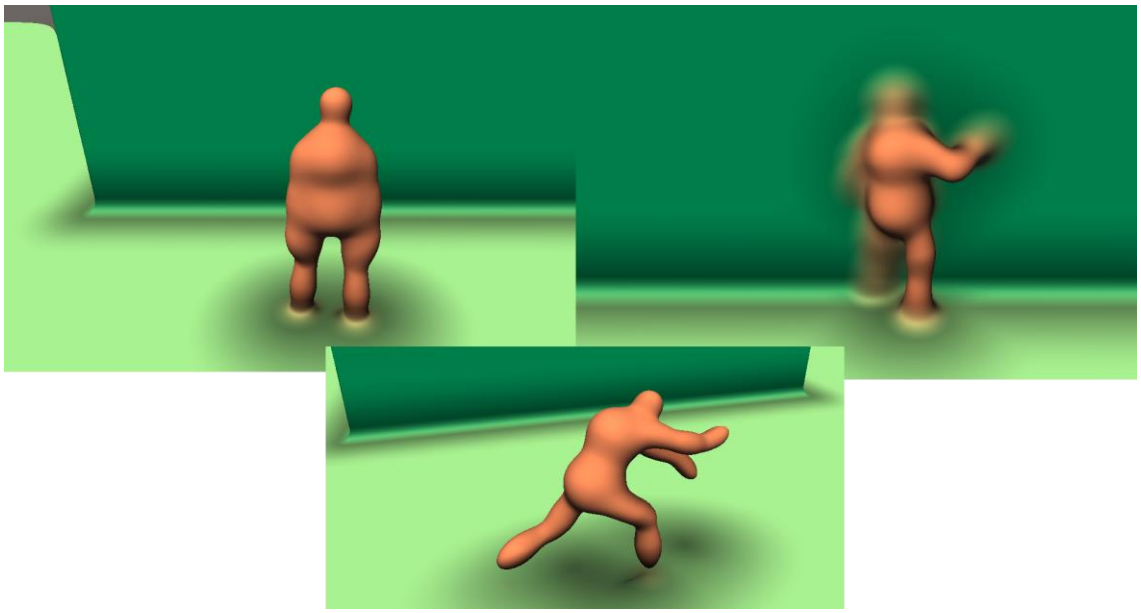


*Figura 44: Escena con un personaje renderizado con funciones de distancia*

## 4.4 Tests

Estas escenas han servido de pruebas del sistema de raymarching de funciones de distancia. El ordenador donde se han hecho las pruebas tiene las siguientes especificaciones:

| | |
|---|---|
| CPU | Intel Core i7-5820K @ 3.30 GHz (12 cores) |
| RAM | 16295 MB |
| GPU | NVIDIA GeForce GTX 1070 |
| VRAM | 8088 MB. Max texture size 16384 px. Shader level 50. |
| Screen | 1920x1080 @ 60Hz |
| Graphics API | Direct3D 11 |
| OS | Windows 10 64 bits |

Los resultados de los experimentos pueden verse a continuación. En las tablas se comparan las escenas a diferentes resoluciones usando distintos algoritmos de iluminación y sombreado y se miden los milisegundos que tardan en renderizarse. El algoritmo de sombras usado es el de sombras suaves y la iluminación se ha realizado con una luz direccional siguiendo el modelo de iluminación de Blinn-Phong con brillos especulares. El número de primitivas de cada escena es:

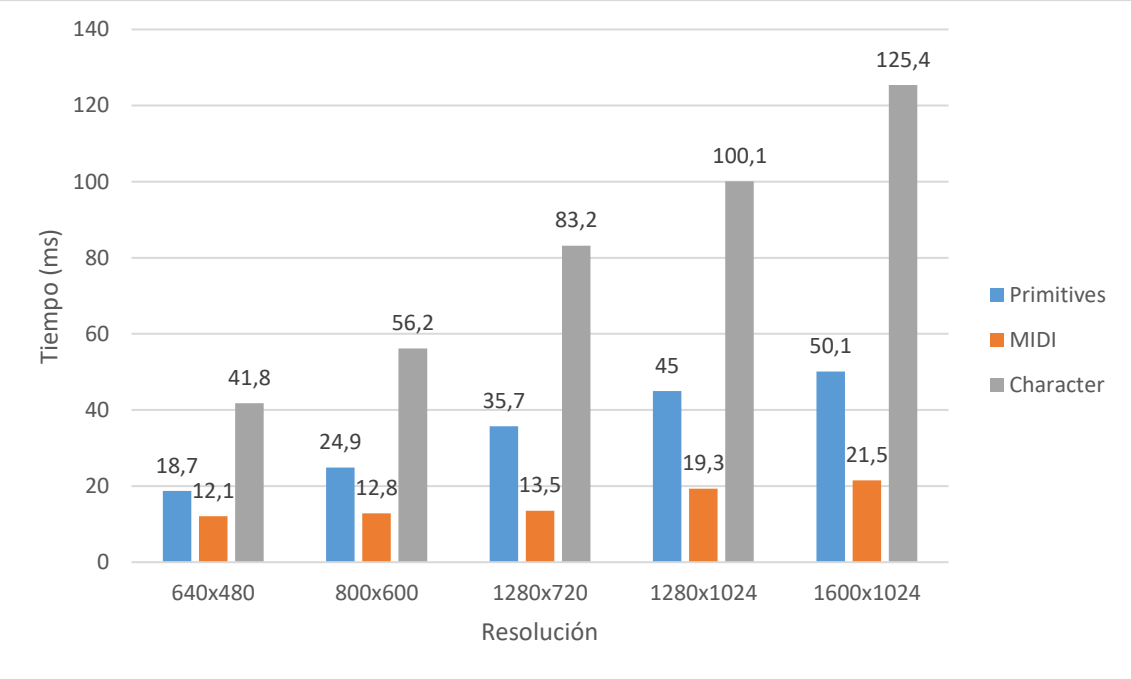
| Escena | Número de primitivas |
|---|---|
| Primitives | 6 |
| MIDI | 5 |
| Character | 15 |

### 4.4.1 Test 1

- Sombras suaves.
- Oclusión ambiental.
- Iluminación especular.

La siguiente tabla muestra los tiempos promedio de renderizado en ms.

| Escena/Resolución | 640x480 | 800x600 | 1280x720 | 1280x1024 | 1600x1024 |
|---|---|---|---|---|---|
| Primitives | 18,7 | 24,9 | 35,7 | 45,0 | 50,1 |
| MIDI | 12,1 | 12,8 | 13,5 | 19,3 | 21,5 |
| Character | 41,8 | 56,2 | 83,2 | 100,1 | 125,4 |



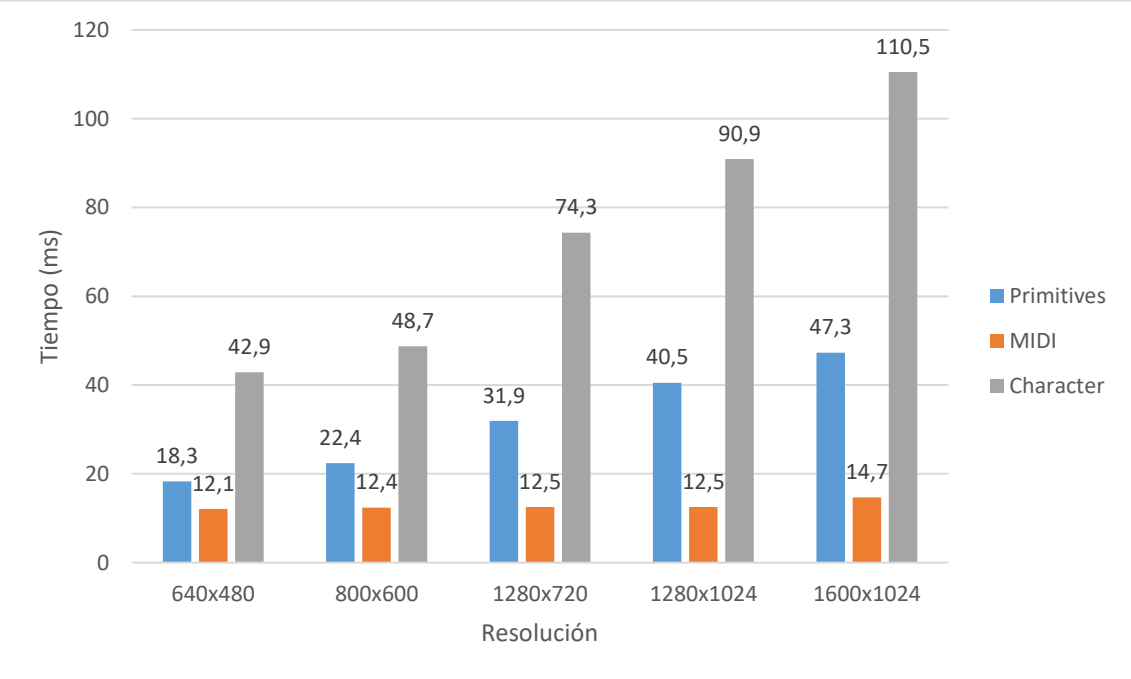

La escena que menos tiempo tarda en renderizarse es la MIDI, ya que contiene un número menor de primitivas, sin embargo la diferencia principal con la escena Primitives es que esta tiene entre sus primitivas un plano (el suelo de la escena), que ocupa toda la pantalla y, por tanto, todo el buffer donde se renderiza. La escena Character presenta un sistema más complejo donde las primitivas de tipo esfera se están mezclando entre ellas y el entorno. Esto hace que el coste temporal de renderizar se dispare. Para resolver el cuello de botella del número de primitivas y hacer el sistema lo más independiente posible de esta variable se tendrán que usar estructuras de datos espaciales que filtren las primitivas para evaluar menos funciones de distancia en cada punto del espacio.

### 4.4.2 Test 2

- Sombras suaves.
- Oclusión ambiental.
  - Sin iluminación especular.

La siguiente tabla muestra los tiempos promedio de renderizado en ms.

| Escena/Resolución | 640x480 | 800x600 | 1280x720 | 1280x1024 | 1600x1024 |
|---|---|---|---|---|---|
| Primitives | 18,3 | 22,4 | 31,9 | 40,5 | 47,3 |
| MIDI | 12,1 | 12,4 | 12,5 | 12,5 | 14,7 |
| Character | 42,9 | 48,7 | 74,3 | 90,9 | 110,5 |

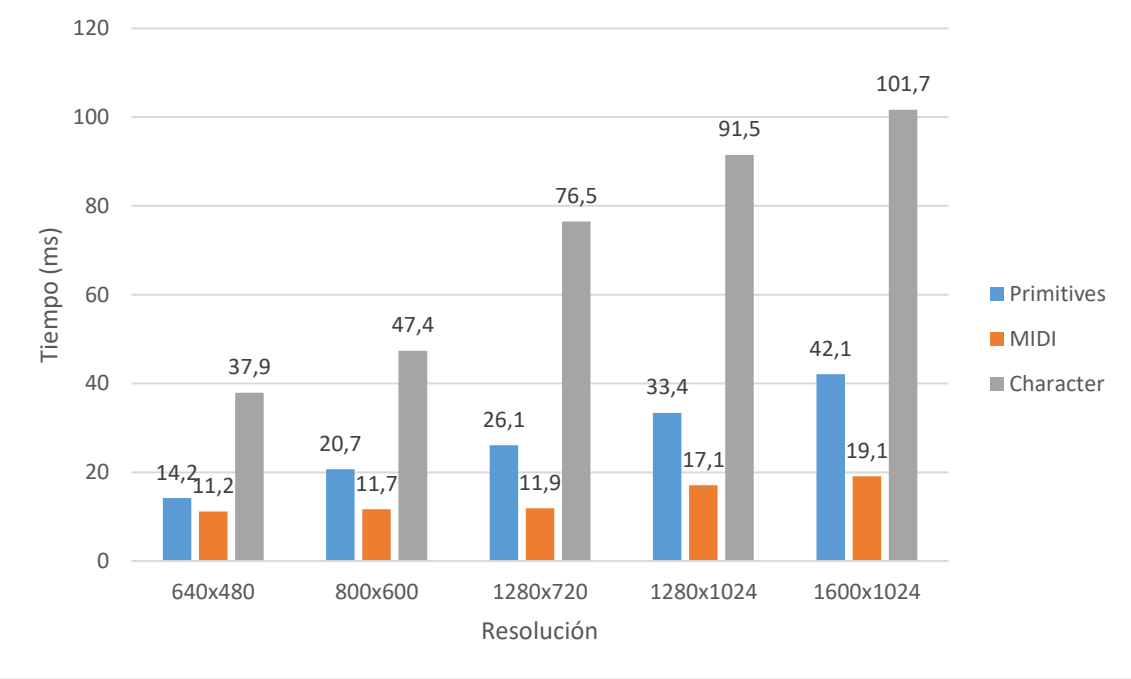


### 4.4.3 Test 3

- Sombras suaves.
  - Sin oclusión ambiental.
  - Sin iluminación especular.

La siguiente tabla muestra los tiempos promedio de renderizado en ms.

| Escena/Resolución | 640x480 | 800x600 | 1280x720 | 1280x1024 | 1600x1024 |
|---|---|---|---|---|---|
| Primitives | 14,2 | 20,7 | 26,1 | 33,4 | 42,1 |
| MIDI | 11,2 | 11,7 | 11,9 | 17,1 | 19,1 |
| Character | 37,9 | 47,4 | 76,5 | 91,5 | 101,7 |

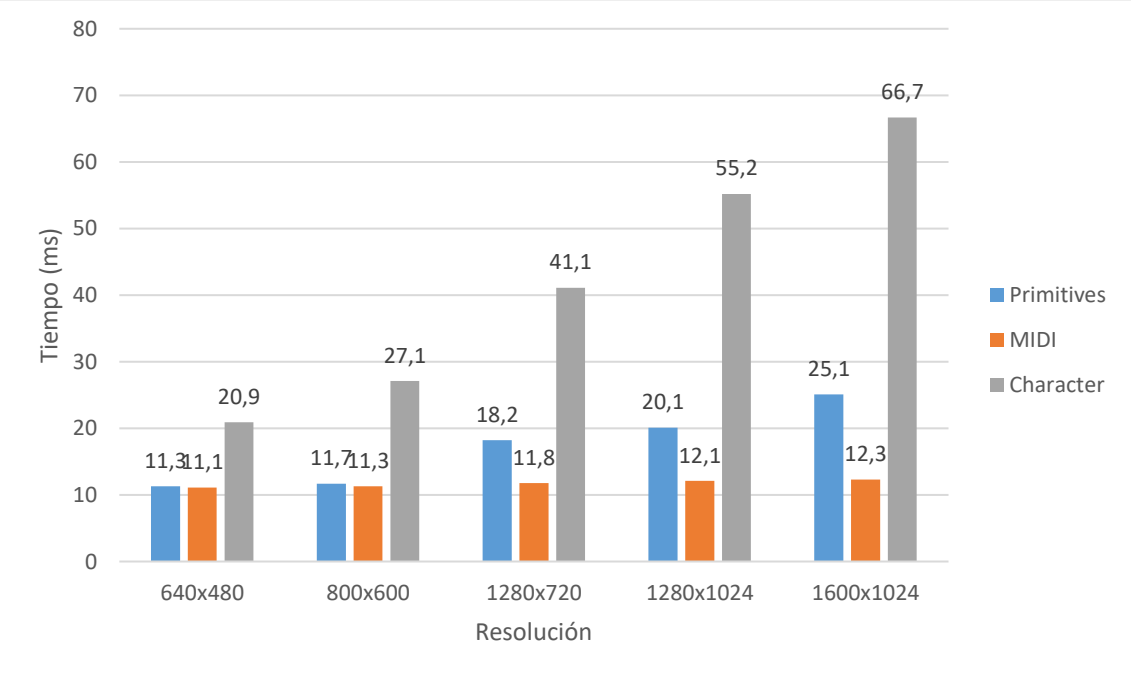


### 4.4.1 Test 4

- Sin sombras suaves.
- Sin oclusión ambiental.
- Sin iluminación especular.

La siguiente tabla muestra los tiempos promedio de renderizado en ms.

| Escena/Resolución | 640x480 | 800x600 | 1280x720 | 1280x1024 | 1600x1024 |
|---|---|---|---|---|---|
| Primitives | 11,3 | 11,7 | 18,2 | 20,1 | 25,1 |
| MIDI | 11,1 | 11,3 | 11,8 | 12,1 | 12,3 |
| Character | 20,9 | 27,1 | 41,1 | 55,2 | 66,7 |




Dados los resultados se observa que el algoritmo más costoso es el cálculo de las sombras. Esto tiene sentido, ya que para calcularlas se realiza otra marcha sobre el vector de iluminación L, coste que se incrementaría en función del número de luces de la escena.

# 5. Conclusiones y trabajo futuro

Las funciones de distancia son una herramienta matemática muy versátil para distintos campos de la computación gráfica, como el renderizado de volúmenes, cálculos de sombreado en función de la distancia o el renderizado de fuentes. El algoritmo que se usa para renderizarlas, raymarching, depende de la resolución del buffer donde se renderiza y actualmente es costoso de ejecutar incluso en GPUs modernas.

Hace una década técnicas como el trazado de rayos o *raytracing* no se podían usar en aplicaciones en tiempo real. En la actualidad existen soluciones mixtas de rasterización y raytracing que consiguen tiempos interactivos. En unos años, con la mejora del hardware, será posible que aplicaciones interactivas más complejas puedan usar raymarching para renderizar volúmenes.

El trabajo realizado ha consistido en implementar un sistema de renderizado de funciones de distancia mediante raymarching para el motor Unity. El sistema se ha implementado como un Unity package que puede ser importado en proyectos propios donde se pueden usar tanto mallas de polígonos como primitivas renderizadas con funciones de distancia. El paquete puede descargarse desde el repositorio de GitHub https://github.com/aquarterofpixel/Raymarching y cuenta con un manual de uso.

Como trabajo futuro, con el objetivo de mejorar la implementación del sistema de renderizado de funciones de distancia mediante raymarching se proponen las siguientes características o puntos críticos a abordar:

- Implementar un sistema de iluminación siguiendo el modelo **physical-based rendering** (PBR).
- Añadir un sistema de capas para mezclar las primitivas usando los operadores de unión, intersección y resta.
- Añadir primitivas más complejas.
- Añadir más operadores de desplazamiento, distorsiones y deformaciones.
- Optimizar el algoritmo de raymarching marchando sobre un subconjunto de rayos más pequeño que el total de la resolución y haciendo un interpolado de píxeles. Los rayos menos importantes se encuentran en el exterior de la pantalla.
- Explorar el uso de los nuevos shaders de raytracing que proporcionan las GPUs modernas.

# Anexo

## Repository documentation

### Raymarching

This repository contains a toolkit to render signed distance functions with raymarching in Unity.

#### Installation

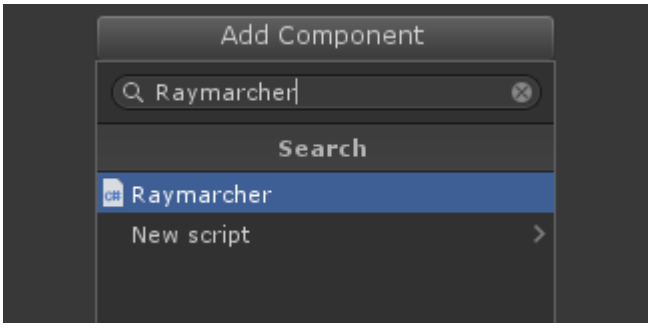
The toolkit is structured as a Unity package. To install it in your Unity project follow the next steps:

1. Clone the repository on your machine.

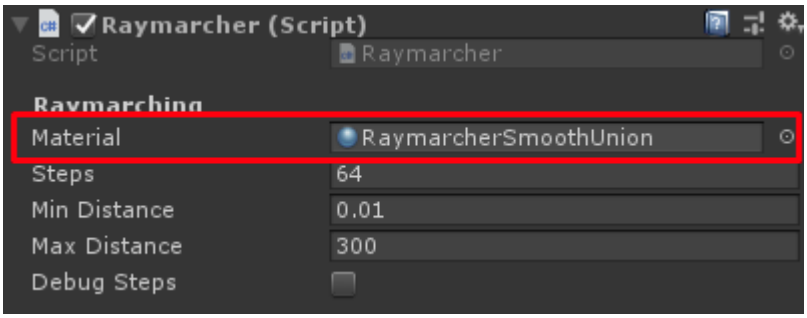
2. Open the file `MyUnityProject/Packages/manifest.json` and add the dependency of the cloned package `"com.aquarterofpixel.raymarching": "file:path/to/Raymarching"`.

#### How to use it

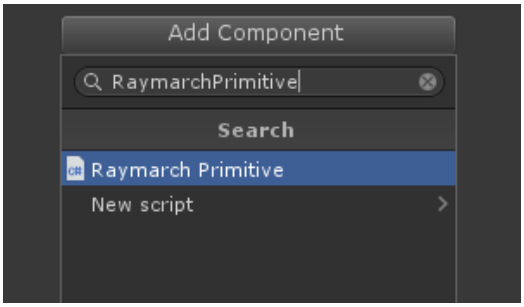
1. Attach a `Raymarcher` component to the camera.




2. Select the material used to render. Two materials are included in the package, `RaymarcherUnion` and `RaymarcherSmoothUnion`.

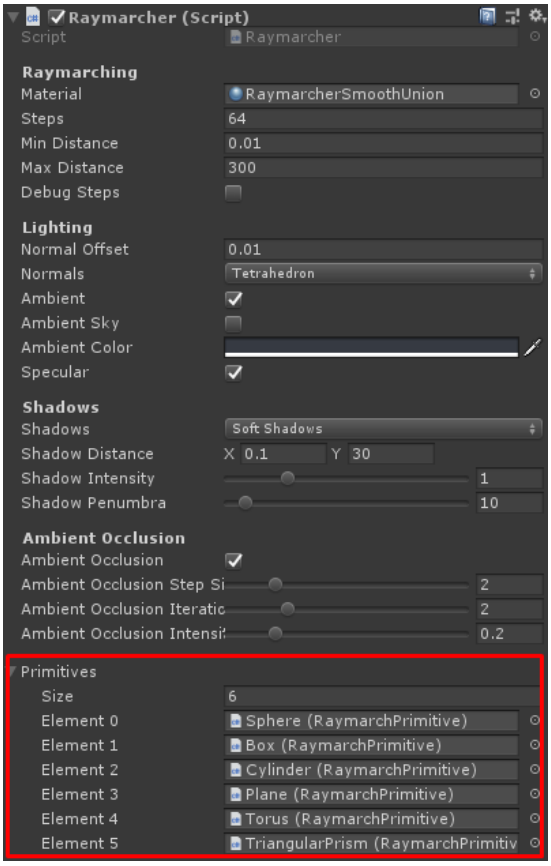


3. Add a `RaymarchPrimitive` component to a gameobject.




4. Select the parameters and material used to render the primitive.


5. Add the `RaymarchPrimitive` component to the primitives list of the `Raymarcher` component.




Now you can move, rotate or scale the raymarch primitive with the Unity transform tools.

## Raymarcher shaders

It is also posible to implement a more complex interaction between primitives creating a raymarcher shader. To create a raymarcher shader go to the context menu `Create > Raymarcher Shader`. The shader contains the function `RaymarchData SignedDistanceField(float3 position)` that is executed in the fragment shader. The function does the following calculations:

1. Iterates over the primitives array.
2. Transforms the target position from world space to the primitive local space.
3. Calculates the distance using the signed distance function of the primitive at the transformed position.
4. The union operator `data.distance = opUnion(data.distance, d)` is applied to the primitives. Other operators can be found in the files SignedDistanceOperations.hlsl and SignedDistanceBooleanOperations.hlsl.
5. Returns the data of the closest primitive.

## Signed distance functions

Plane

```
// n: Normal to the plane (normalized).
float sdPlane( float3 p, float4 n )
{
	return dot(p, n.xyz) + n.w;
}
```

Box

```
// l: Length.
float sdBox( float3 p, float3 l )
{
	float3 d = abs(p) - l;
	return length(max(d, 0.0))
		+ min(max(d.x, max(d.y, d.z)), 0.0);
}
```

Sphere

```
// r: Radius.
float sdSphere( float3 p, float r )
{
	return length(p) - r;
}
```

Ellipsoid

```
// r: Radius.
float sdEllipsoid( float3 p, float3 r )
{
	float k0 = length(p / r);
	float k1 = length(p / (r * r));
	return k0 * (k0 - 1.0) / k1;
}
```

Cylinder

```
// h: Height.
// r: Radius.
float sdCylinder( float3 p, float h, float r )
{
	float2 d = abs(float2(length(p.xz), p.y)) - float2(r, h);
	return min(max(d.x, d.y), 0.0) + length(max(d, 0.0));
}
```

Capsule

```
// h: Height.
// r: Radius.
float sdCapsule( float3 p, float h, float r )
{
	p.y -= clamp(p.y, -h / 2.0, h / 2.0);
	return length(p) - r;
}
```

Torus

```
// d: Diameter.
// t: Thickness.
float sdTorus( float3 p, float d, float t )
{
	float2 q = float2(length(p.xz) - d, p.y);
	return length(q) - t;
}
```

Triangular Prism

```
// h: Height.
// w: Width.
float sdTriangularPrism( float3 p, float h, float w )
{
	float3 q = abs(p);
	return max(q.z - w, max(q.x * 0.866025 + p.y * 0.5, -p.y) - h * 0.5);
}
```

Hexagonal Prism

```
// h: Height.
// w: Width.
float sdHexagonalPrism( float3 p, float h, float w )
{
	const float3 k = float3(-0.8660254, 0.5, 0.57735);
	p = abs(p);
	p.xy -= 2.0 * min(dot(k.xy, p.xy), 0.0) * k.xy;
	float2 d = float2(
		length(p.xy - float2(clamp(p.x, -k.z * h, k.z * h), h)) *
sign(p.y - h),
		p.z - w);

	return min(max(d.x, d.y), 0.0) + length(max(d, 0.0));
}
```

## Signed distance operations

Round

```
// Round a shape.
// Subtracts some distance (jumping to a different isosurface).
// If you want to preserve the overal volume of the shape, shrink the source
primitive by the same amount you are rounding it.
// r: Radius.
float opRound( float d, float r )
{
	return d - r;
}
```

Onion

```
// Carve the interior or give thickness to primitive.
// Use it multiple times to create concentric layers.
// t: Thickness.
float opOnion( float d, float t )
{
	return abs(d) - t;
}
```

Twist X

```
// Twist the point across the x axis.
// k: Twist factor.
float3 opTwistX( float3 p, float k )
{
	float c = cos(k * p.x);
	float s = sin(k * p.x);
	float3x3 m = float3x3(1, 0, 0, 0, c, -s, 0, s, c);
	float3 q = mul(m, p);
	return q;
}
```

Twist Y

```
// Twist the point across the y axis.
// k: Twist factor.
float3 opTwistY( float3 p, float k )
{
	float c = cos(k * p.y);
	float s = sin(k * p.y);
	float3x3 m = float3x3(c, 0, s, 0, 1, 0, -s, 0, c);
	float3 q = mul(m, p);
	return q;
}
```

Twist Z

```
// Twist the point across the z axis.
// k: Twist factor.
float3 opTwistZ( float3 p, float k )
{
	float c = cos(k * p.z);
	float s = sin(k * p.z);
	float3x3 m = float3x3(c, -s, 0, s, c, 0, 0, 0, 1);
	float3 q = mul(m, p);
	return q;
}
```

Mod 1

```
// Repeat space along one axis and returns the cell (optional).
// float c = pMod1(p.x, 1);
// c: Cell.
float opMod1( inout float p, float size )
{
	float halfsize = size * 0.5;
	float c = floor((p + halfsize) / size);
	p = fmod(p + halfsize, size) - halfsize;
	p = fmod(-p + halfsize, size) - halfsize;
	return c;
}
```

Mod Positive 1

```
// Repeat space along one positive axis and returns the cell (optional).
// float c = pModPositive1(p.x, 1);
// c: Cell.
float opModPositive1( inout float p, float size )
{
	float halfsize = size * 0.5;
	float c = floor((p + halfsize) / size);
	if( p >= 0 )
	{
		p = mod(p + halfsize, size) - halfsize;
	}

	return c;
}
```

Mod Interval 1

```
// Repeat space along one axis from start to end cells and returns the cell
(optional).
// float c = pModInterval1(p.x, 1, 0, 10);
// c: Cell.
float opModInterval1( inout float p, float size, int start, int end )
{
	float halfsize = size * 0.5;
	float c = floor((p + halfsize) / size);
	p = mod(p + halfsize, size) - halfsize;
	if( c > end )
	{
		p += size * (c - end);
		c = end;
	}
	if( c < start )
	{
		p += size * (c - start);
		c = start;
	}

	return c;
}
```

Mod 2

```
// Repeat space along two axes and returns the cell (optional).
// float2 c = pMod2(p.x, p.y, 1);
// c: Cell.
float2 opMod2( inout float p1, inout float p2, float2 size )
{
	float2 c = floor((float2(p1, p2) + size * 0.5) / size);
	p1 = mod(p1 + size.x * 0.5, size.x) - size.x * 0.5;
	p2 = mod(p2 + size.y * 0.5, size.y) - size.y * 0.5;
	return c;
}
```

Mod 3

```
// Repeat space along three axes and returns the cell (optional).
// float3 c = pMod3(p.x, p.y, p.z, 1);
// c: Cell.
float3 opMod3( inout float p1, inout float p2, inout float p3, float3 size )
{
	float3 c = floor((float3(p1, p2, p3) + size * 0.5) / size);
	p1 = mod(p1 + size.x * 0.5, size.x) - size.x * 0.5;
	p2 = mod(p2 + size.y * 0.5, size.y) - size.y * 0.5;
	p3 = mod(p3 + size.z * 0.5, size.z) - size.z * 0.5;
	return c;
}
```

## Signed distance boolean operations

Smooth minimum polynomial

```
// Smooth min polynomial.
// Not commutative, order dependent.
// Suffers from second order discontinuities.
// k: Controls the radius/distance of the smoothness (k = 0.1).
float smin_pol( float a, float b, float k )
{
	float h = clamp(0.5 + 0.5 * (b - a) / k, 0, 1);
	return lerp(b, a, h) - k * h * (1 - h);
}
```

Smooth minimum polynomial color

```
float smin_pol_color( float a, float b, float k, float4 c1, float4 c2, out
float4 c )
{
	float h = clamp(0.5 + 0.5 * (b - a) / k, 0, 1);
	c = lerp(c2, c1, h);
	return lerp(b, a, h) - k * h * (1 - h);
}
```

Smooth minimum polynomial cubic

```
// Smooth min polynomial cubic.
// Not commutative, order dependent.
// Has second order continuity, important for preventing lighting artifacts.
// k: Controls the radius/distance of the smoothness (k = 0.1).
float smin_cubic( float a, float b, float k )
{
	float h = max(k - abs(a - b), 0) / k;
	return min(a, b) - h * h * h * k * (1.0 / 6.0);
}
```

Smooth minimum power

```
// Smooth min power.
// Generalize to more than two distances.
// k: Controls the radius/distance of the smoothness (k = 8).
float smin_pow( float a, float b, float k )
{
	a = pow(a, k); b = pow(b, k);
	return pow((a*b) / (a + b), 1.0 / k);
}
```

Smooth minimum exponencial

```
// Smooth min exponencial.
// Generalize to more than two distances.
// Commutative when taken in pairs, smin(a, smin(b, c)) is equal to smin(b,
smin(a, c)).
// k: Controls the radius/distance of the smoothness (k = 32).
float smin_exp( float a, float b, float k )
{
	float res = exp2(-k * a) + exp2(-k * b);
	return -log2(res) / k;
}
```

Smooth maximum polynomial

```
// Smooth max polynomial.
// Not commutative, order dependent.
// Suffers from second order discontinuities.
// k: Controls the radius/distance of the smoothness (k = 32).
float smax_pol( float a, float b, float k )
{
	float h = clamp(0.5 - 0.5 * (b - a) / k, 0, 1);
	return lerp(b, a, h) + k * h * (1 - h);
}
```

Smooth maximum polynomial color

```
float smax_pol_color( float a, float b, float k, float4 c1, float4 c2, out
float4 c )
{
	float h = clamp(0.5 - 0.5 * (b - a) / k, 0, 1);
	c = lerp(c2, c1, h);
	return lerp(b, a, h) + k * h * (1 - h);
}
```

Union

```
// Union boolean operation.
float opUnion( float d1, float d2 )
{
	return min(d1, d2);
}
```

Intersection

```
// Intersection boolean operation.
float opIntersection( float d1, float d2 )
{
	return max(d1, d2);
}
```

Subtraction

```
// Subtraction boolean operation.
// Not commutative and depending on the order of the operands it will produce
different results.
float opSubtraction( float d1, float d2 )
{
	return max(-d1, d2);
}
```

Smooth union

```
// Union boolean operation.
// k: Controls the radius/distance of the smoothness (k = 0.1).
float opSmoothUnion( float d1, float d2, float k )
{
	return smin_pol(d1, d2, k);
}
```

Smooth union color

```
float opSmoothUnionColor( float d1, float d2, float k, float4 c1, float4 c2,
out float4 c )
{
	return smin_pol_color(d1, d2, k, c1, c2, c);
}
```

Smooth intersection

```
// Intersection boolean operation.
// k: Controls the radius/distance of the smoothness (k = 0.1).
float opSmoothIntersection( float d1, float d2, float k )
{
	return smax_pol(d1, d2, k);
}
```

Smooth intersection color

```
float opSmoothIntersectionColor( float d1, float d2, float k, float4 c1, float4
c2, out float4 c )
{
	return smax_pol_color(d1, d2, k, c1, c2, c);
}
```

Smooth subtraction

```
// Subtraction boolean operation.
// Not commutative and depending on the order of the operands it will produce
different results.
// k: Controls the radius/distance of the smoothness (k = 0.1).
float opSmoothSubtraction( float d1, float d2, float k )
{
	return smax_pol(-d1, d2, k);
}
```

Smooth subtraction color

```
float opSmoothSubtractionColor( float d1, float d2, float k, float4 c1, float4
c2, out float4 c )
{
	return smax_pol_color(-d1, d2, k, c1, c2, c);
}
```

# Referencias bibliográficas

## Publicaciones y artículos


| | |
|---|---|
| [1] | Quílez, Íñigo. Modeling with distance functions. 2008. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm |
| [2] | Quílez, Íñigo. Raymarching Distance Fields. 2017. https://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm |
| [3] | Quílez, Íñigo. Numerical normals for SDFs. 2018. https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm |
| [4] | Quílez, Íñigo. Smooth Minimum. 2013. https://www.iquilezles.org/www/articles/smin/smin.htm |
| [5] | Quílez, Íñigo. Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes. NVScene 2008. http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf |
| [6] | Quílez, Íñigo. Free Penumbra Shadows for Raymarching Distance Fields. 2010. https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm |
| [7] | Green, Chris. Valve. SIGGRAPH 2007. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. https://steamcdn-a.akamaihd.net/apps/valve/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf |
| [8] | Wright, Daniel. Dynamic Occlusion with Signed Distance Fields. SIGGRAPH 2015. http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/DistanceFieldAmbientOcclusion |
| [9] | Aaltonen, Sebastian. GPU-based clay simulation and ray-tracing tech in Claybook. GDC 2018. https://twvideo01.ubm-us.net/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf |

## Libros


| | |
|---|---|
| [10] | Real-Time Rendering, Fourth Edition.<br>http://www.realtimerendering.com/ |
| [11] | NVIDIA GPU Gems.<br>https://developer.nvidia.com/gpugems/GPUGems/ |


## Sitios web


| | |
|---|---|
| [12] | SIGGRAPH.<br>https://www.siggraph.org/ |
| [13] | Game Developers Conference.<br>https://www.gdconf.com/ |
| [14] | Shadertoy.<br>https://www.shadertoy.com/<br>https://en.wikipedia.org/wiki/Shadertoy |
| [15] | Unity.<br>https://unity.com/ |
| [16] | Unreal Engine.<br>https://www.unrealengine.com/ |
| [17] | Claybook.<br>https://www.claybookgame.com/ |
| [18] | TextMesh Pro.<br>http://digitalnativestudios.com/ |

## Demoscenes


| | |
|---|---|
| [19] | Demoscene.info<br>http://www.demoscene.info/the-demoscene/ |
| [20] | Demoscene Awards.<br>http://awards.scene.org/index.php |
| [21] | Revision.<br>https://en.wikipedia.org/wiki/Revision_(demoparty) |
| [22] | NVScene, Nvision.<br>https://en.wikipedia.org/wiki/Nvision |
| [23] | Quílez, Íñigo.<br>https://www.shadertoy.com/user/iq |
| [24] | Malin, Paul.<br>https://www.shadertoy.com/user/P_Malin |
| [25] | Rgba, Elevated.<br>http://www.pouet.net/prod.php?which=52938 |
| [26] | Quite & Orange, Cdak.<br>http://www.pouet.net/prod.php?which=55758 |
| [27] | Quílez, Íñigo. Arlo. 2015.<br>https://www.shadertoy.com/view/4dtGWM |
| [28] | Quílez, Íñigo. Snail. 2015.<br>https://www.shadertoy.com/view/ld3Gz2 |
| [29] | Quílez, Íñigo. Ladybug. 2017.<br>https://www.shadertoy.com/view/4tByz3 |