TecnoCampus
Escola Superior
Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Grau en Enginyeria Informàtica de Gestió i Sistemes d'Informació

GPU ACCELERATED RAY TRACER WITH SAH-BVH

Memòria

MIQUEL VALVERDE PARERA
TUTOR: DR. DAVID RÓDENAS PICÓ

CURS ACADÈMIC 2020-2021

TecnoCampus
Mataró-Maresme

Abstract

This project is focused on studying a rendering technique known as Ray Tracing. The state of the art is deeply analysed, and the same technique is putted in practice, by using those resources that demonstrate to cause greater optimization impact. A both hardware (GPU) and software (SAH-BVH) accelerated Ray Tracer is implemented. Finally, we compare the obtained results from an efficiency point of view. We can assert that both GPU and BVH are crucial for Ray Tracing in today's consumer hardware. Still, Ray Tracing auge is far from possible at its fullest.

Resum

L'objectiu d'aquest treball és l'estudi de la tècnica de síntesi d'imatge coneguda com Ray Tracing. S'analitza a fons l'estat de l'art i es posa en pràctica la mateixa tècnica, fent ús dels recursos que demostren causar el major impacte en la seva optimització. S'implementa un Ray Tracer accelerat per hardware (GPU) i per software (SAH-BVH) alhora. Finalment es comparen els resultats obtinguts a nivell d'eficiència. Podem asserir que tant la GPU com el BVH són elements crucials pel traçat de rajos a temps real a dia d'avui en hardware de consum. No obstant, l'auge del traçat de rajos està lluny de ser viable en la seva màxima expressió.

Resumen

El objetivo de este trabajo es el estudio de la técnica de síntesis de imagen conocida como Ray Tracing. Se analiza a fondo el estado del arte y se pone en práctica la misma técnica, haciendo uso de los recursos que demuestran causar el mayor impacto en su optimización. Se implementa un Ray Tracer acelerado por hardware (GPU) y por software (SAH-BVH) a la vez. Finalmente se comparan los resultados obtenidos a nivel de eficiencia. Podemos afirmar que tanto la GPU como el BVH son elementos cruciales para el trazado de rayos a tiempo real hoy en día en hardware de consumo. No obstante, el auge del trazado de rayos está lejos de ser viable en su máxima expresión.

Índex

Índex de figures

Índex de taules

Glossari de termes

AABB	Volum mínim contenidor d'un objecte alineat als eixos.	*Axis Aligned Bounding Box*
BRDF	Funció que determina com reflexa la llum en una superfície.	*Bidirectional Reflectance Distribution Function*
BVH	Estructura espacial de volums jeràrquics per accelerar consultes.	*Bounding Box Hierarchy*
CUDA	Es pot referir a l'arquitectura de *Nvidia* o la plataforma de desenvolupament sota el paradigma de programació en paral·lel.	*Compute Unified Device Architecture*
CGI	Gràfics i visualitzacions generats per computador.	*Computer Generated Imagery*
GPU	Processador extern dedicat al processament d'imatges.	*Graphics Processing Unit*
GPGPU	GPU amb funcionalitats ampliades per fer còmputs de propòsit general.	*General Purpose Graphics Processing Unit*
IDE	Entorn integrat que conté eines per facilitar la programació.	*Integrated Develoment Environment*
Offline Render	Render que no és a temps real.	
Ray Tracing	Algorisme de renderitzat centrat en la càmera.	
Rendering	Procés de síntesis d'imatges	(traducció de l'autor: renderitzat)
Rasterization	Algorisme de renderitzat centrat a l'objecte.	(traducció de l'autor: rasterització)
SAH	Heurístic per construir BVHs.	*Surface Area Heuristic*

1 Introducció

A dia d'avui, els videojocs, pel·lícules, visualitzacions arquitectòniques i simulacions mèdiques, consisteixen en imatges generades per computador (CGI). Segons el seu nivell d'interacció, és crucial que la velocitat de resposta sigui a temps real.

La renderització a temps real és una característica essencial pel funcionament d'un videojoc de forma interactiva. Alhora, per complaure als consumidors amb productes versemblants, és imprescindible produir imatges fotorealistes [1]. Malauradament, una característica suposa un *tradeoff* amb l'altre.

Fins ara, han conviscut dues tècniques per generar imatges: la Rasterització i el *Ray Tracing*. Si només es busca l'eficiència sense contemplar el fotorealisme, és òptim aplicar la *Rasterització*. Mentre que si busca el fotorealisme, la solució és el *Ray Tracing*.

Per tal que la *Rasterització* imiti el fotorealisme, mantenint la seva eficiència, l'algorisme s'ha refinat enginyosament fins arribar als seus límits. Malauradament, per molt que es refini mai serà capaç de simular alguns fenòmens i comportaments lumínics.

Ray Tracing és una tècnica elegant de síntesis d'imatges que simula el comportament de la llum. Fins ara, únicament s'ha utilitzat el *Ray Tracing* en camps que no precisaven de la renderització a temps real, com a la indústria cinematogràfica.

Per contra, el *Ray Tracing* presenta un repte computacional per a que es pugui dur a terme a temps real. No ha estat fins al 2018 que s'ha començat a aplicar *Ray Tracing* per renderitzar videojocs, on és crucial la interacció a temps real.

El fotorealisme a temps real es considera un camp de recerca a dia d'avui encara. A l'actualitat recent però, s'han establert els primers treballs que poden generar fotorealisme a temps real aplicant *Ray Tracing* amb hardware de consum.

Aquest treball es focalitza en investigar les tècniques principals per accelerar el temps de l'algorisme. Es tenen en compte, i s'implementen, alguns mecanismes contemplats per l'estat de l'art de la pròpia tècnica on es demostra la millora de temps del *Ray Tracing*.

Per últim, amb intenció de restringir l'objectiu del treball, cal mencionar que el treball només es basa en estudiar i implementar els mecanismes per a poder aplicar *Ray Tracing* de forma eficient, i no té per objectiu simular el comportament dels rebots secundaris de la llum.

2 Marc teòric

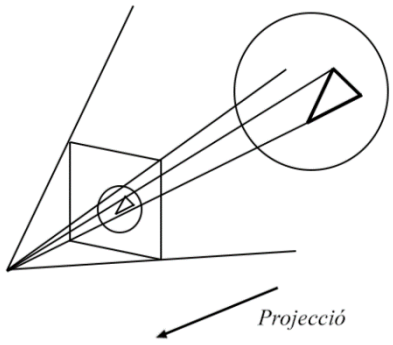
Existeixen dues tècniques fonamentals per a la síntesi d'imatges: la rasterització i el *ray tracing*. En aquest epígraf es mostren teories existents i s'aprofundeix en el *ray tracing*.

2.1 Context i antecedents

2.1.1 Rasterization

El concepte de rasterització aplicat a gràfics, és un algorisme de síntesis d'imatge centrat a l'objecte. Es basa en recórrer la geometria i aplicar una projecció planar en espai de píxels. Aquest principi s'ha utilitzat fins a l'actualitat, i probablement s'utilitzi al futur en molts contextos degut a la seva eficiència, que presenta una complexitat lineal $O(n)$.

Figura 1. Principi teòric de la Rasterització



Font: Elaboració pròpia

Es dibuixen els objectes primitius des de darrera fins al davant, essent els del davant que tapen els de darrere. Una forma de resoldre el problema de la visibilitat es coneix com l'Algoritme del Pintor, on la ordenació dels objectes és relativa a la vista de la càmera. Malauradament presenta una sèrie de problemes quan dos triangles fan intersecció, ja que no és capaç de determinar quin fragment del triangle està per davant o per darrere.

Les APIs de gràfics no implementen aquest mecanisme per les problemàtiques exposades. És més comú aplicar *Z-Buffering* [2]. La ordenació amb un *Z-Buffer* consisteix en prendre la profunditat de la coordenada de la interpolació baricèntrica z_j (en cas que es tracti d'un triangle) i comparar-la amb el valor actual del *buffer* z_i. Si la profunditat trobada és menor

$z_i > z_j$ (el número és major), es substitueix el valor del *buffer* $z_i \rightarrow z_j$. Finalment només es pinten els píxels que queden per davant.

Un dels primers rasteritzadors va aparèixer cap al 1960 a *Bell Labs*, per Michael Noll [3].

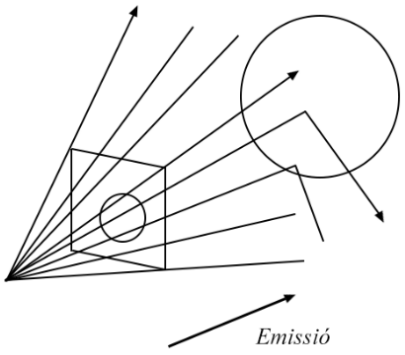
2.1.2 Ray Tracing

Simplicitat, paral·lelisme i accessibilitat són els termes que venen a la ment quan es pensa en *Ray Tracing*. Avui en dia no hi ha dubte que és el vehicle definitiu per assolir fotorealisme.

El concepte de traçat de rajos prové del segle XVI [4] sense cap connotació computacional, sinó com a tècnica artística. La primera connotació computacional sobre l'objecte d'estudi al 1968, s'anomena *Ray Casting* [5].

Fins fa dos anys, al 2018, aquesta tècnica no és a temps real en hardware de consum degut al cost elevat per a sintetitzar imatges, i s'anomena *Path Tracing* [6]. El qual és una extensió del model proposat per Appel [5] per a generar un model d'il·luminació fotorealista de base física [7].

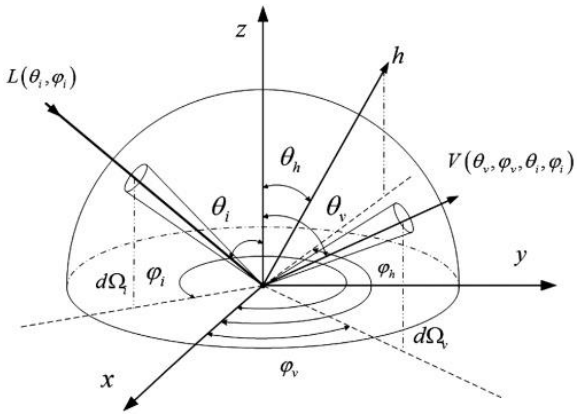
Figura 2. Principi teòric del Ray Tracing



Font: Elaboració pròpia

La idea del *Ray Tracing* parteix de que una font de llum emet fotons en una o més direccions. Aquesta ona contínua de fotons es desplaça per l'aire fins que col·lisiona en una superfície. En base a les característiques físiques de la superfície, la direcció dels fotons es pot veure afectada de diferents maneres, veure Figura 3.

Figura 3. BRDF



Font: [8]

BRDF és una funció per calcular la dispersió de la llum que compleix el principi de reciprocitat. Garanteix que si s'inverteix el sentit de la llum incident i reflectida, el valor de BRDF es manté invariant. Gràcies al principi de reciprocitat es pot resoldre el següent problema, del qual neix la idea del *Ray Tracing*.

Si els rajos s'originen des de la font de llum, esdevé un problema costós, sinó impossible, saber quins acaben entrant a la lent de la càmera. És per això que s'inverteix tot el procés, i els rajos s'originen des de la càmera. D'aquesta manera només es calcula la radiància visible i necessària. *Ray Tracing* dona resposta a la radiància entrant a cada píxel de la càmera.

2.1.3 Ray Tracing vs Rasterization

Ray Tracing és un algorisme de síntesis d'imatge centrat a la càmera, a diferència de la rasterització centrada en l'objecte. Aquest aspecte fonamental defineix els dos mecanismes principals de síntesis d'imatges, explica que l'operació que la rasterització no pot proveir és l'habilitat de conèixer què hi ha al voltant d'un punt. Essent tant essencial, que la majoria de tècniques per rasteritzar són solucions enginyoses per superar aquesta limitació.

Ray Tracing és capaç de superar aquestes limitacions. Simula el comportament de la llum i genera un model d'il·luminació físicament acurat. Alhora és un algorisme que simula diversos fenòmens de llum com ombres, reflexions i refractivitat, entre d'altres efectes de forma versàtil.

Tot i així, la rasterització supera en eficiència computacional al traçat de rajos, la qual cosa suposa una contrapartida. En aquest TFG s'analitza com sobrepassar aquesta contrapartida.

2.2 Estat de l'Art

En l'actualitat 2020, l'estat de l'art es troba molt més avançat mantenint l'essència del concepte del traçat de rajos original, i és factible a temps real sota algunes consideracions que s'exposen en aquesta secció.

A la conferència de SIGGRAPH 2018, Jensen Huang (CEO i fundador de *Nvidia*) presenta, segons ell, la major innovació en gràfics per ordinador en més d'una dècada. Anunciant la nova arquitectura Turing [9].

Figura 4. Arquitectura de Nvidia Pascal vs Turing

Font: Nvidia, SIGGRAPH 2018

La nova arquitectura Turing inclou tres mòduls, dels quals ens interessen dos; *Shader/Compute Core* i *RT Core*.

Els *Shader/Compute Cores* són el que a l'arquitectura Pascal engloben tota la GPU per satisfer funcionalitats de còmput genèriques. A la nova arquitectura Turing es manté aquest mòdul amb alguna millora respecte l'anterior.

Els *RT Cores* són un fragment de la GPU dedicat exclusivament a resoldre consultes de Ray Tracing. Aquestes consultes són analitzades més endavant, i s'explica la seva necessitat dins de l'algorisme per al *Ray Tracing*. El que ens interessa saber ara, és que les consultes resolen una part molt costosa del *Ray Tracing*. És per això que *Nvidia* ha invertit més d'una dècada en

aquesta tecnologia, i fins fa dos anys no ha sortit a producció. En aquest treball només s'utilitza la part computacional de la GPU per resoldre les consultes mencionades, que no els *RT Cores*.

Aquest nou mòdul és necessari perquè estandarditza les APIs de gràfics afegint cada una de les fases del algorisme *Ray Tracing*. Fins ara les GPUs només estaven estandarditzades amb la rasterització. Avui en dia els *RT Cores* no són accessibles de cara el programador. Només s'hi pot accedir a través de APIs de forma indirecte.

Perquè el *Ray Tracing* sigui factible a temps real, no n'hi ha prou amb els *RT Cores* en l'arquitectura Turing, i la seva estandardització en APIs gràfiques.

S'utilitza una tècnica d'escalat de resolució d'imatges [10] basada en intel·ligència artificial per renderitzar a menys resolució. Aquest factor augmenta la taxa de fotogrames per segon, llevat el cost d'alteracions subtils a la imatge final. *Nvidia* és propietària d'una tècnica anomenada *Deep Learning Super Sampling* (DLSS 2.0) [11].

Per últim i no menys important, també és possible gràcies a l'ús de tècniques per a detectar patrons de punts i dissimular el soroll provocat per interrompre l'algorisme, el qual pot requerir molt de temps en convergir. Aquesta tècnica és basada en filtrat bilateral i s'anomena *Denoising* [12].

Totes aquestes tècniques que conformen l'estat de l'art del traçat de rajos, s'apliquen majoritàriament a la indústria de videojocs. Malgrat, són vigents tècniques de *offline rendering* perquè obtenen resultats físicament acurats, sense que el temps esdevingui un factor crucial, com és a la indústria cinematogràfica.

2.3 Estructures d'Acceleració Espacial

Un accelerador espacial és una estructura de dades que optimitza consultes mitjançant la subdivisió de l'espai. La subdivisió de l'espai fragmenta un problema per disminuir la seva complexitat a l'hora de processar-lo.

2.3.1 Necessitat

La implementació naïf del traçat de rajos mostra un problema potencialment costós de resoldre a nivell computacional. Aquest problema implica que:

Figura 5. Pseudocodi de Ray Tracing

Algoritme: Pseudocodi de Ray Tracing

```
1: for all pixel in camera do
2:     Generació de raig
3:     for all triangle in escena do
4:         Intersecció triangle-raig
5:     end for
6: end for
```

Font: Elaboració pròpia amb LaTeX

La complexitat de temps que descriu aquest algorisme és lineal $O(n * m)$. Es considera una complexitat molt elevada si augmenta el nombre de píxels o bé el nombre de triangles.

A continuació es mostra un exemple del cost potencial. Suposant que es vol renderitzar una escena de trenta mil triangles, (en un videojoc és un pressupost de polígons baix per a un dispositiu mòbil) a 1080p (resolució estàndard d'una pantalla que conté 2,073,600 píxels), equival a seixanta mil milions de càlculs a resoldre, per generar un sol fotograma. On cada càlcul és una intersecció triangle-raig. Aquesta situació només té en compte els *Primary Rays*, però si afegim els rebots secundaris de la llum, pot arribar a un número de computacions infinitament elevat. De tal manera que si volem renderitzar a 60 FPS, on cada fotograma pot trigar un màxim de 0.016 segons, resulta un algorisme inviable a l'actualitat.

Per assegurar la viabilitat del *Ray Tracing* en temps real, s'utilitzen algorismes per a subdividir l'espai i reduir la complexitat esmentada.

2.3.2 Estructures Convencionals

Els algorismes per a fer recorreguts espacials i accelerar el traçat de rajos poden intercanviar-se segons les necessitats de l'escena que es renderitza. Els més utilitzats són *Uniform Grid*, *Octree*, *KDTree* i BVH.

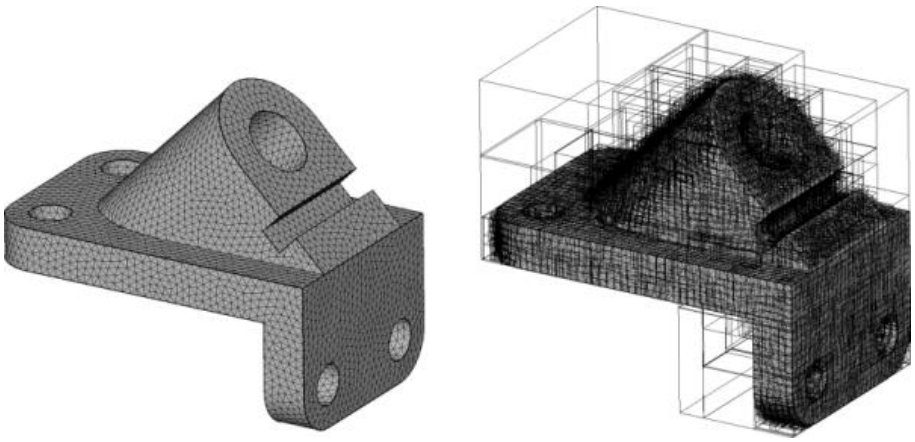
Tal com es mostra a l'article [13], BVH demostra la major versatilitat i rendiment superant a tota la resta. Ha estat escollit per a *Nvidia* a la seva arquitectura Turing amb RT Cores [9].

D'aquesta manera BVH s'ha convertit en l'algorisme estàndard de l'actualitat per al traçat de rajos.

2.3.3 Bounding Volume Hierarchy (BVH)

El BVH és una estructura de dades per emmagatzemar i iterar sobre un conjunt d'objectes geomètrics. Tots els objectes geomètrics estan continguts en volums AABB que formen els nodes fulla de l'arbre. Aquests nodes estan agrupats en conjunts majors, formant una jerarquia subdivisions que s'arranja en una estructura arbòria, veure gràficament a la Figura 6.

Figura 6. Conjunt de triangles amb construcció de BVH

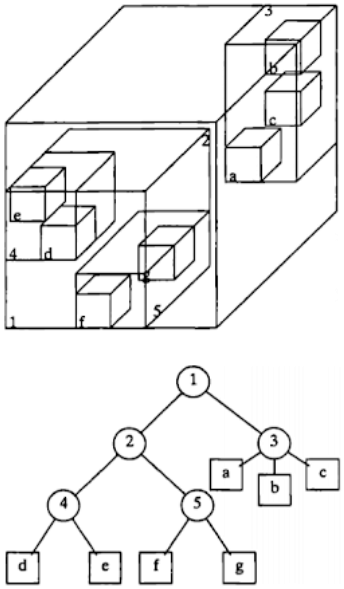


Font: Imatge de CGAL

Les jerarquies de volums són utilitzades per fer consultes eficients i donar a conèixer la posició d'un objecte a l'espai. En aquest treball, esdevé una estructura que optimitza les interseccions entre triangles i rajos. El BVH no solament té aquesta finalitat, també s'utilitza el BVH per detectar col·lisions entre objectes dispersos arbitràriament a l'espai.

BVH presenta un **set disjunt d'objectes**, a diferència de les altres estructures convencionals mencionades, que presenten un **set disjunt de regions**. Aquest fet permet realitzar comprovacions d'interseccions una única vegada per regió.

Figura 7. Representació esquemàtica de BVH



Font: Transparències de Princeton

A la figura Figura 7, els números encerclats representen nodes interns. Els nodes interns contenen una o més referències a altres nodes interns, o bé a nodes fulla. Les lletres requadrades representen nodes fulla. Els nodes fulla contenen referències als objectes, que poden ser triangles.

Aquesta estructura redueix dramàticament la complexitat de temps per al traçat de rajos, amb una complexitat logarítmica $O(log\ n)$ sense contemplar el pitjor dels casos.

2.3.4 Surface Area Heuristic (SAH)

Per a construir un BVH existeixen diversos heurístics que no són objecte d'estudi en aquest TFG. L'objecte d'estudi però, es centra únicament en SAH el qual va ser publicat al 1990 en aquest article [14]. Tal com explica l'article, s'observa que el nombre de raigs que tendeixen a intersecar amb un objecte, és proporcional a la superfície de l'àrea d'aquell objecte.

SAH assumeix que la probabilitat de que un raig produeixi intersecció amb un volum, és igual a la superfície de l'àrea, tal que:

$$C(TA,TB) = P(A)\sum_{i=1}^{N_{VA}} c_{isct}(VA_i) + P(B)\sum_{i=1}^{N_{VA}} c_{isct}(VB_i)$$

On $C(TA,TB)$ és el cost que provoca Tallar el Volum A i el Volum B per a un eix específic.

On $P(VA)$ i $P(VB)$ són les probabilitats de intersecció amb el Volum A i el Volum B.

On N_{VA} i N_{VB} són el Número d'objectes que contenen el Volum A i el Volum B.

On VA_i i VB_i són l'ièssim objecte que contenen el Volum A i el Volum B.

On c_{isct} equival al cost d'una intersecció raig-triangle, o un objecte arbitrari.

SAH millora el recorregut en estructures espacials basades en jerarquies de volums. L'heurístic estableix una relació inversament proporcional de la superfície de l'àrea en vers a la densitat poligonal, que disminueix la probabilitat que un raig produeixi intersecció amb un volum, i procedeixi a calcular la intersecció amb els nodes interns.

La superfície de l'àrea SA_V d'un volum es pot calcular de diferents formes. Tenint en compte que s'utilitzen volums paral·lepípedes, podem calcular $SA_V = Vx + Vy + Vz$. No és necessari calcular la superfície real degut a que aquest valor s'utilitza com a un heurístic. És més ràpid sumar les tres components axials.

En un article escrit per I. Wald [15], actual Director de *Ray Tracing* a *Nvidia*, es detalla com construir SAH-BVH de forma eficient.

2.4 Intersecció Raig-Triangle

En aquest apartat s'analitza la part nuclear de les computacions que fins ara s'han considerat abstractes a la complexitat de temps dels algoritmes. Per entendre com es calcula la intersecció entre Raig-Triangle, cal definir:

- Un raig R format per un punt O, una direcció D i un paràmetre escalar t.
- Un triangle $T \in R^3$ format per tres punts A, B i C.

Existeixen diferents formes de computar geomètricament si es produeix una intersecció Raig-Triangle. En aquest TFG s'aplica la tècnica proposada per Möller-Trumbore al 2005 en aquest article [16], ja que ha esdevingut una de les tècniques més ràpides i eficients.

Figura 8. Pseudocodi de Möller-Trumbore

Algoritme: Pseudocodi de Möller-Trumbore

1: **Inputs:**
   $R(o, d)$;
   $T(a, b, c)$;
2: **Initialize:**
   $ab \leftarrow Tb - Ta$
   $ac \leftarrow Tc - Ta$
   $cr \leftarrow Rd \times ac$
   $det \leftarrow Rd \cdot cr$
3: **if** $det < \varepsilon$ **then return** Termini sense intersecció
4: **end if**
5: $s \leftarrow Ro - Ta$
6: $invDet \leftarrow 1/det$
7: $u \leftarrow s \cdot cr \cdot invDet$
8: **if** $u < 0 || u > 1$ **then return** Termini sense intersecció
9: **end if**
10: $qv \leftarrow s \times invDet$
11: $v \leftarrow Rd \cdot qv \cdot invDet$
12: **if** $v < 0 || u + v > 1$ **then return** Termini sense intersecció
13: **end if**
14: $t \leftarrow ca \cdot qv \cdot invDet$
15: **if** $t > \varepsilon$ **then return** Termini amb intersecció
16: **end ifreturn** Termini sense intersecció

Font: Elaboració pròpia amb LaTeX basada en l'algorisme existent

Com es pot veure a la Figura 8, s'utilitza èpsilon ε per establir un llindar a la perpendicularitat del raig i el triangle. D'aquesta manera s'eviten les interseccions no deterministes. Aquest llindar, o tolerància, pot ser escollit per l'usuari. S'utilitza $\varepsilon = 10^{-6}$ en base a l'estàndard aritmètic de coma flotant $IEEE_{745}$ de 32-bit. Si s'utilitza la doble precisió, aquest llindar pot ser més proper a zero.

Aquest mètode és eficient perquè ràpidament desvia el flux de lògica cap a un resultat binari, que indica si hi ha intersecció o no hi ha intersecció. D'aquesta manera es pot fer una comprovació molt ràpida entre un raig i molts triangles que no causen intersecció, perquè el resultat és descartat.

2.5 Accelerador amb GPU

2.5.1 Necessitat

Un dels elements centrals que s'estudia en aquest TFG, és el mòdul de hardware que coneixem com a targeta de gràfics. Hem vist que utilitzar *Ray Tracing* pot obligar a realitzar trilions de càlculs, els quals impossibiliten aquesta tècnica de forma interactiva.

Degut a la naturalesa de l'algorisme, *Ray Tracing* permet calcular cada píxel independentment a la resta. Aleshores és factible imaginar com es pot encarar el problema de forma solapada en paral·lel, per comptes resoldre'l seqüencialment.

Aquesta és l'especialitat de la GPU, per la qual cosa podem accelerar la velocitat de càlculs a través d'aquest hardware. Degut a la seva naturalesa, la GPU s'emmarca a la categoria SIMD de la taxonomia de Flynn [17].

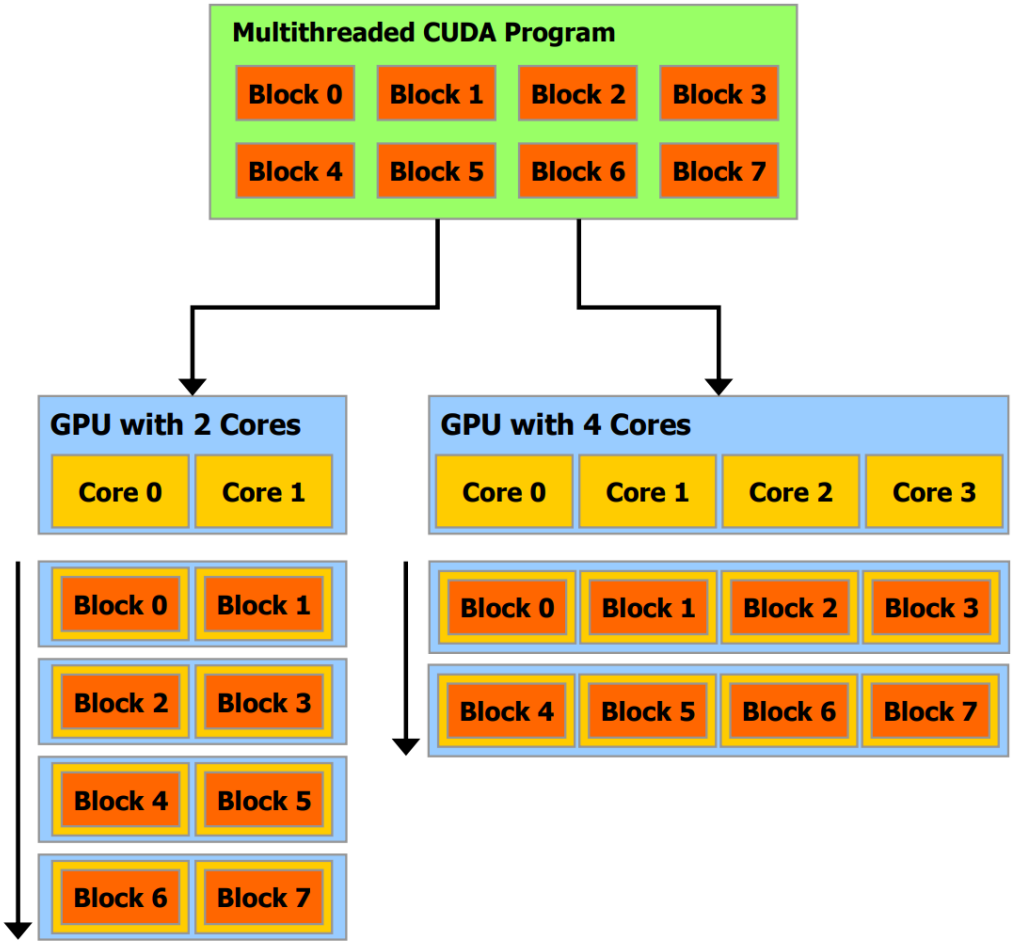
2.5.2 Arquitectura CUDA i gestió de memòria

En aquest treball, ens centrem específicament amb el model de memòria de l'arquitectura CUDA de *Nvidia*, ja que és la tecnologia escollida pera l'objecte d'estudi. Alguns dels conceptes però, són extrapolables amb altres fabricants.

En el nucli d'una GPU es diferencien tres abstraccions clau; una jerarquia de fils agrupats en blocs, mecanismes de sincronització i nivells de memòria compartida.

Un programa està dividit en *Blocks* formats per *Threads* que s'executen independentment entre si. Incrementant el número de *Cores* automàticament s'executa el programa en menys temps que una GPU amb menys *Cores* [18]. Per aquesta raó CUDA es considera un model de programació escalable.

Figura 9. Escalabilitat de CUDA



Font: Documentació oficial de CUDA

La GPU està formada per un conjunt de memòries amb els diferents graus d'accessibilitat.

Els *Registers* són la memòria més ràpida, accessible només des dels *Threads*. El número de registres, pot variar segons el model de GPU. Si bé el programa excedeix el límit imposat pel hardware, aleshores passa a utilitzar el següent nivell de memòria, anomenada *Thread-Local-Global Memory*.

La *Thread-Local-Global Memory*, és utilitzada per emmagatzemar variables que excedeixen el tamany dels registres, o bé contenen valors indeterminats en temps de compilació. Aquesta memòria, que porta confusió pel nom, es considera memòria global DRAM. A la Figura 10 es pinta en color taronja per indicar aquesta dualitat *Local-Global*.

Les variables que especifiquin que són compartides, s'emmagatzemaran a *Shared Memory*. Aquesta memòria té un grau d'accessibilitat de tot el *Block* al que pertany. Aquesta memòria permet la comunicació entre *Threads*, ja que estan agrupats a dins de *Blocks*. És més ràpida que la *Thread-Local-Global Memory*.

Per últim, tenim tres altres tipus de memòria global. El grau d'accessibilitat de la memòria global, és per a tots els *Threads* pertanyents a qualsevol *Block.* A pesar de que la *Constant Memory*, *Texture Memory* i *Global Memory* es consideren d'accessibilitat global, tenen petites diferències optimitzades per a l'ús que es pretén donar a cadascuna.

Una GPU està formada per un determinat nombre de *Streaming Multiprocessors* (SMs), que alhora contenen un conjunt de *Thread-Blocks* per SM.

Cada SM està format per múltiples *Stream Processors* (SPs), on cada SP equival a un sol *Thread* amb els seus *Registers* corresponents. Sovint els SPs són referits com a *CUDA Cores*. El número de SPs que cada SM conté, ve determinat per la *Compute Capability* (cc).

També discernim entre nivells de memòries de caixet (*cache levels*) L1 i L2. S'utilitza un L1 per SM, mentre que s'utilitza un L2 a accessible des de múltiples SMs. L1 és utilitzada per la *Shared Memory* essent més ràpida. L2 és més lenta, però tots els accessos a *Global Memory* es realitzen a través de L2.

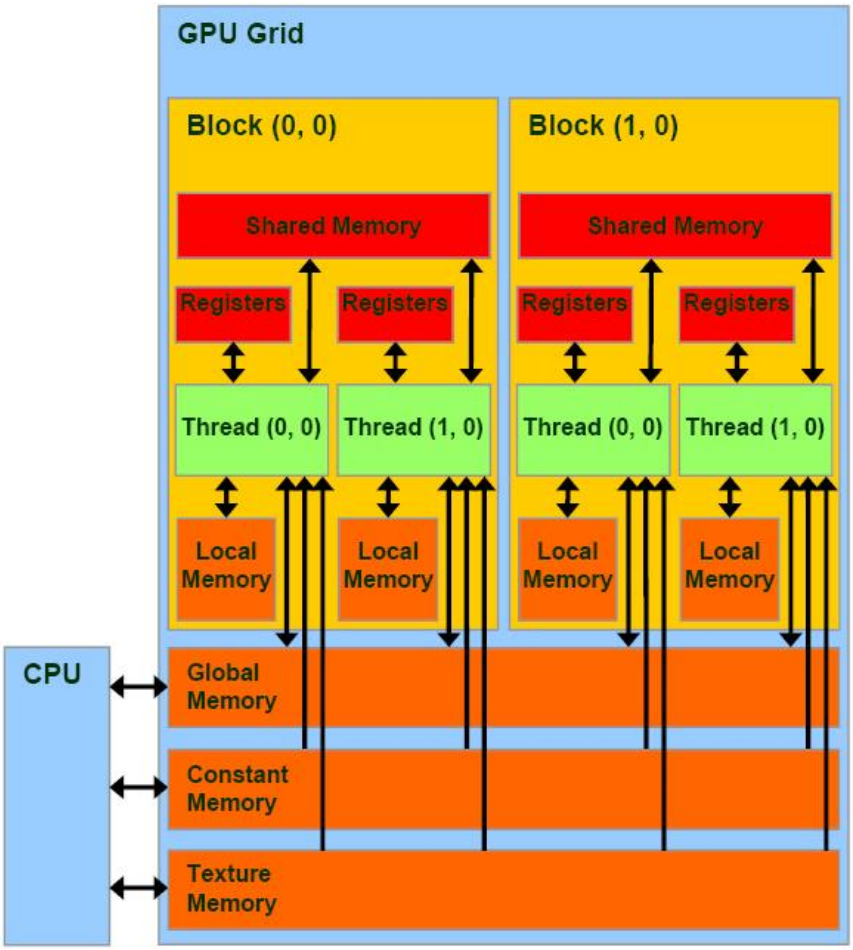
A l'hora d'executar-se, la CPU exerceix una comunicació amb la *Global Memory* per assignar la memòria que serà utilitzada pel *kernel*. CUDA facilita aquest procés, i només cal assignar i alliberar la memòria exacte que serà utilitzada o deixada d'utilitzar respectivament, amb instruccions del tipus *cudaMalloc i cudaMemcpy* [18].

Un altre concepte important que defineix l'execució dels *Threads*, són els *Warps*. Aquests indiquen agrupacions de *Threads* que són executats simultàniament. La majoria de GPUs d'avui en dia tenen *Warps* de 32 *Threads*.

Per dur a terme l'execució dels *Warps* es disposa dels *Warp-Schedulers* i *Warp-Dispatchers* de cada SM. Són els que s'encarreguen de distribuir la feina en els seus respectius SPs i executar els *Threads* de forma concurrent o paral·lela en funció de l'arquitectura.

La Figura 10 representa a nivell esquemàtic tots els conceptes explicats en aquest apartat.

Figura 10. Arquitectura de memòria d'una GPU.



Font: Documentació oficial de CUDA

Com hem pogut veure, el paradigma de programació que ofereix CUDA, és un model paral·lel només apte per a GPUs de *Nvidia*. Aquesta plataforma gestiona la memòria i llença *kernels* que s'executen en paral·lel, des de la CPU.

2.6 Necessitats d'informació

Les necessitats d'informació per a l'objecte d'estudi són àmplies, i engloba tot el contingut vist en el marc teòric i totes les referències a la bibliografia.

La necessitat d'informació principal és respecte la programació en GPU i la pròpia gestió de la memòria.

En segon lloc, l'estudi d'algorismes espacials per a accelerar el còmput d'interseccions també suposa una necessitat d'informació.

També és una necessitat entendre les tècniques de *render* existents per a comprendre l'aportació d'aquest treball.

3 Objectius i abast

L'objectiu és implementar un GPU-AABB-SAH-BVH sense utilitzar cap recurs ni llibreria de tercers: Rebre per entrada un fitxer *Wavefront .obj* i generar per sortida una imatge estàtica que representi el model 3D.

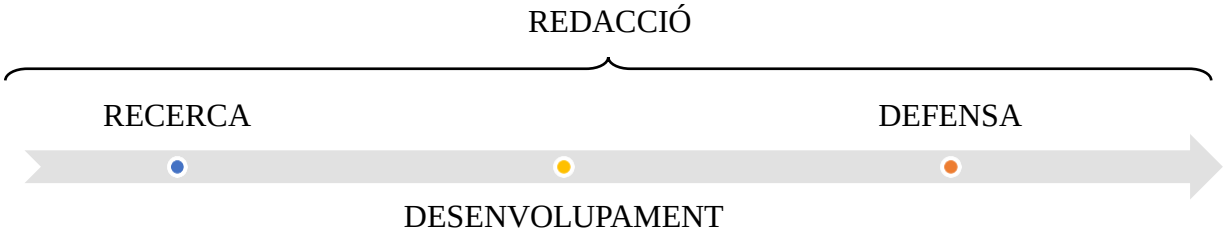
Aquest projecte està composat de tres etapes generals:

Figura 11. Etapes Generals

Font: Elaboració Pròpia

Subjacents a les etapes generals, discernim tres etapes de treball que no necessàriament estan alineades amb les anteriors. A continuació es mostren les fites que delimiten **l'abast del projecte** amb els objectius mesurables corresponents:

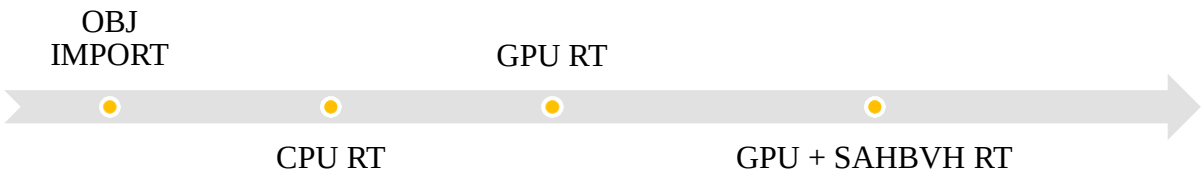
Figura 12. Etapes de Treball, abast del projecte



Font: Elaboració Pròpia

Les fites que formen els objectius del desenvolupament estan descrites a continuació, i delimiten **l'abast del producte**.

Figura 13. Objectius Mesurables, abast del producte



Font: Elaboració Pròpia

Enumeració i descripció detallada dels objectius principals:

- **OBJ IMPORT**: Llegir i importar arxius 3D amb format *.obj*. Les darreres etapes utilitzen aquesta informació per a generar imatges.

- **CPU RT**: Programar un *Ray Tracer* que treballi amb la unitat central de processament. Com a sortida genera una imatge.

- **GPU RT**: Adaptar el *Ray Tracer* creat anteriorment per que treballi amb la targeta de gràfics extraient-ne el seu potencial. Com a sortida genera una imatge més ràpida que l'anterior.

Objectius secundaris:

- **GPU + SAHBVH RT**: Crear un BVH amb l'heurístic SAH que treballi amb la targeta de gràfics. Extreure el potencial a través del hardware i el software. Com a sortida genera una imatge més ràpida que totes les anteriors.

3.1 No objectiu

Avui en dia al món hi ha aproximadament una vintena de *renderers* (en català representador gràfic) destacables en producció molt especialitzats per a les seves respectives tasques. No es pren per objectiu crear un altre *renderer* competent amb totes les característiques ni qualitat de producció. Tampoc es pretén crear un *renderer* a temps real, ja que per a això es necessita implementar un *frame buffer* i un conjunt de recursos de tercers, o interoperabilitat amb altres APIs. No es pretén superar l'eficiència dels *RT Cores*, ja que aquesta nova tecnologia no està ni tan sols al abast del públic (no està exposada per a ser programable) a dia d'avui, com s'explica al marc teòric.

4 Anàlisi de referents

Existeixen molts referents històrics que s'han enfrontat al problema d'optimitzar l'algoritme. *Ray Tracing* presenta molts reptes més enllà de la intersecció massiva entre raigs-triangles, com ara el transport de la radiància. L'anàlisi de referents només es centra en problema d'optimitzar la intersecció raig-triangle.

Existeixen referents que utilitzen principis i tècniques descrites al marc teòric. Entre ells es destaca *Renderman* de *Pixar Animation Studios* i *Frostbite* de *DICE*. A pesar de proposar diferents solucions, tots tenen en comú que tendeixen cap a una mateixa solució que requereix de hardware especialitzat.

Hi ha dues categories de *Ray Tracers* MIMD paral·lels; aquells que utilitzen algorismes basats en *image-space partition* i d'altres que utilitzen *object-space partition*. Si s'utilitza *imatge-space partition* hi ha un sol processador dedicat a una regió de la imatge.

Distribuir la feina amb *image-space partition* implica que hi ha d'haver memòria compartida entre tots els processadors, o bé que s'acoblin els resultats obtinguts de cada processador quan finalitza el procés. Per imatges complexes, sovint produïdes per estudis referents d'animació, el cost de comunicació entre els processadors per ensamblar els fragments de cada imatge és negligible. Aquesta solució és factible en hardware comercial bàsic, *Pixar* ho va utilitzar a la producció de la pel·lícula *Toy Story*.

Aquest model no és aplicable a un videojoc com ara *Battlefield V* de *Electronic Arts*, on un mil·lisegons esdevé la diferència entre viure o morir, i es necessita processament a temps real. Per escenes senzilles l'ample de banda i l'ús de memòria és excessiu, és més adequat utilitzar *object-space partition*, on cada processador és responsable de calcular intersecció amb un grup de triangles.

Avui en dia ningú es qüestiona si l'arquitectura SIMD és la més adient per la problemàtica descrita, i sovint el *Ray Tracing* en GPU es banalitza i s'etiqueta com una problema trivial de resoldre. Tanmateix, la comunitat de gràfics ha dubtat més d'un cop si SIMD és la millor solució al problema [19]. Sorgeixen reptes a l'hora de concatenar el llançament de rajos produïts per la divergència de *threads* de la GPU. També sorgeixen reptes alhora de gestionar

la memòria. Aquest fenomen pot dificultar l'eficiència del *Ray Tracing* a pesar que sigui una arquitectura SIMD.

Nvidia ha optat per modificar el hardware afegint una canonada ASIC dedicada, que connecta directament a la *Shared Memory* amb un accés directe al BVH que conté la *object-space partition*. Aquesta tecnologia es coneix com RTX. Fins avui a l'actualitat (2020), aquesta solució ha demostrat els millors resultats a nivell d'eficiència.

El videojoc *Battlefield V* de *Electronic Arts* (2018) es considera el primer referent en aplicar la tècnica RTX de *Nvidia* a nivell de producció. És difícil saber si l'èxit comercial del videojoc va ser per la novetat, o bé pel propi joc. En qualsevol cas, és una fita històrica que ha obert pas a més videojocs de cara el futur que apliquen *Ray Tracing*.

5 Metodologia

Es contemplen diferents etapes que composen la metodologia del projecte:

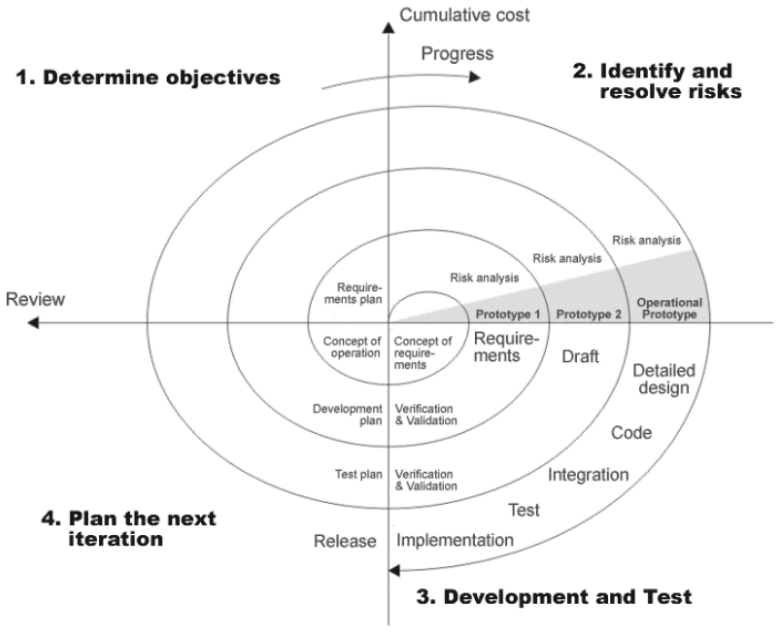
- La primera etapa es basa en la recerca de l'estat de l'art, la qual està inclosa dins l'avantprojecte, en conjunt amb la planificació del desenvolupament del producte.

- La següent consisteix en el desenvolupament del producte en base a l'etapa prèvia. Aquesta segona etapa està dividida entre les cinc fites descrites als Objectius i Abast. Per cada fita es pretén millorar el temps de *rendering*, això determinarà que es pugui avançar a la següent fita, o s'hagi d'iterar sobre la mateixa sense permetre avançar. Per tant el mapa que ens tracem per a l'elaboració d'aquest projecte implica una millora de rapidesa per cada fita assolida.

- La tercera i última etapa és la preparació de la defensa davant del tribunal docent, presentació final de la memòria, i dipòsit de la mateixa.

El pla per autogestionar l'organització i garantir la realització del projecte consisteix en un sistema iteratiu de producció, aplicant el Model en Espiral [20].

Figura 14. Cicle de vida en espiral



Font: Model proposat per Bohem

Aquest model consisteix en iterar donant voltes en espiral, on cada volta equival a una funcionalitat acabada, i en aquest TFG equival a cada fita del producte. Alhora cada fita equival a un subproducte avaluable individualment.

La Taula 1 relaciona les iteracions amb els seus respectius objectius del desenvolupament.

Taula 1. Objectius per iteració

Iteració	**Objectiu**
1	Lectura d'arxius *.obj* i estructura per emmagatzemar objectes.
2	Implementació naïf d'un *Ray Tracer* amb CPU.
3	Implementació d'un *Ray Tracer* amb GPU.
4	Implementació d'un *Ray Tracer* amb GPU i SAH-BVH.

Font: Elaboració pròpia

6 Desenvolupament

6.1 Requeriments del projecte

La col·lecció de requeriments i compromisos del producte final es mostra a continuació. D'aquesta manera es pot provar que s'ha complert els requeriments com a eina de comprovació.

6.1.1 Requeriments funcionals

Pel client final els requeriments funcionals impliquen l'obtenció d'una imatge resultant, amb la millora de la rapidesa de *rendering*.

La següent taula està composada dels requeriments funcionals del producte. S'ha assignat l'identificador en ordre de prioritat. També s'ha assignat l'Estimació *Agile* de Fibonacci per quantificar l'esforç esperat que comporta la complexitat de la tasca.

Per últim, s'ha inclòs un conjunt de requeriments no desitjats, que es consideren fora del domini del treball. S'han inclòs per remarcar quin és el territori que engloba el treball i quin no. S'ha escollit aquests requeriments i no uns altres perquè en un desenvolupament d'escala major, probablement serien els següents passos a seguir, bo i que no els últims.

Taula 2. Requeriments funcionals

ID	Requeriment	Desitjat	Estimació
1	Lectura de arxiu *.obj*	Si	2
2	Camp de visió ajustable de la càmera.	Si	3
3	Traçat de rajos amb CPU.	Si	8
4	Paralelització amb GPU.	Si	8
5	Construcció de SAH-BVH.	Si	13
6	Paralelització amb GPU i SAH-BVH en conjunt.	Si	21
7	Materials BRDF	No	-
8	Il·luminació Global o altres tipus d'il·luminació	No	-

Font: Elaboració Pròpia

Descripció dels requeriments funcionals en format de llista:

1. Capacitar el programa per a rebre un arxiu *.obj* i extraure la informació de triangles únicament.

2. Programar una càmera amb un camp de visió ajustable a través d'un angle, que evoqui l'efecte de perspectiva.

3. Programar un *Ray Tracer* que utilitza la CPU per a generar una imatge, en base a la lectura prèvia dels triangles.

4. Utilitzar la GPU per processar el traçat de rajos programat a l'etapa prèvia.

5. Construir una estructura de dades BVH, amb l'heurístic SAH.

6. Traspassar l'estructura de dades anterior a la GPU, per a que el traçat de rajos sigui simultàniament processat amb GPU i SAH-BVH.

6.1.2 Requeriments tecnològics

A continuació es mostren els recursos tècnics necessaris per a desenvolupar el producte.

- Es necessita una plataforma o SDK per poder programar utilitzant la GPU.
- Un IDE que faciliti les tasques de depuració i la programació en aquesta plataforma escollida.
- Llenguatge de baix nivell per a gestionar la memòria de forma òptima.
- Targeta gràfica de propòsit general (GPGPU), capaç de realitzar tasques de computació.
- Sistema operatiu compatible amb tota la tecnologia esmentada.
- Font de recursos per obtenir models en 3D.

6.2 Desenvolupament del projecte

El desenvolupament del projecte s'aborda a través d'iteracions que alhora equivalen a les funcionalitats escollides del producte final. L'estructura dels següents subapartats comença amb l'especificació de la iteració. Tot seguit s'explica el disseny amb el qual s'aborda la implementació. Per últim, es valora si s'ha aconseguit l'objectiu de la iteració.

6.2.1 *Tech Stack*

Previ al desenvolupament s'escull el conjunt d'eines que s'utilitza a les següents iteracions. L'elecció de les eines es basa en els requeriments tecnològics i es composa de la següent tecnologia:

- Per aplicar *GPU-Threading* s'utilitza *CUDA Toolkit 10.2* de *Nvidia*.
- Com a IDE s'utilitza *Microsoft Visual Studio Community 2019*.
- Com a llenguatge principal s'utilitza *ISO C++ 11 Standard*.
- Targeta gràfica amb arquitectura CUDA, *Quadro K2200*.
- Sistema operatiu *Windows 10 Pro* de 64-bit.
- Font de models de *Stanford 3D Scanning Repository*.

El motiu per el qual s'utilitza CUDA, és l'adaptabilitat amb una GPU *Nvidia*. A pesar d'existir altres opcions universals com OpenCL, els resultats a nivell d'eficiència afavoreixen a CUDA. L'especialització proporciona extreure més partit a un hardware concret, sense donar suport a hardware de *ATI Technologies*.

S'escull la gama *Quadro*, perquè està orientada a estacions de treball i perquè és la GPU que tenim a disposició. A diferència de la gama *GeForce*, que només conté el mode WDDM per a mostrar display de pantalla, *Quadro* pot funcionar en mode TCC (*Tesla Compute Cluster*) i WDDM (*Windows Display Driver Model*) permetent enfocar la GPU a tasques de computació exclusivament. Aquesta versatilitat també aporta funcionalitats extres de *debugging*.

L'IDE *Microsoft Visual Studio* està escollit en base a la bona integració amb la tecnologia descrita. També aporta extensions com *Nsight* per inspeccionar visualment la memòria utilitzada als *kernels* de la GPU.

El llenguatge utilitzat és C/C++ degut a que el compilador MSVC (*Visual C++*, cl.exe) és el suportat per NVCC (*Nvidia Cuda Compiler*) a Windows.

Finalment, com a font de recursos d'objectes 3D, s'utilitza el repositori de models de *Stanford.* La font compleix amb les necessitats del projecte perquè és oberta al públic de forma gratuïta.

6.2.2 Primera iteració

6.2.2.1 Especificació

La primera iteració afronta el primer requeriment funcional descrit (ID: 1). Consisteix en la lectura d'arxius *Wavefront .obj*, més concretament de la informació referent a la malla tessel·lada amb triangles.

Es considera conclòs aquest apartat si temps d'execució, es pot fer la lectura i interpretar un arxiu *.obj*. Aquesta informació ha de ser recuperada en les posteriors etapes de desenvolupament de forma senzilla.

És també un requeriment que no s'utilitzi l'emmagatzematge d'objectes complexos, o provinents d'altres llibreries. Aquest requeriment és una previsió de riscs en futures etapes que utilitzen la GPU, on es vol assegurar la compatibilitat del codi que es desenvolupa en tot moment.

A continuació s'exposa el Taula 3. Cas d'Ús general. Es considera "general" perquè la única diferència respecte les darreres iteracions, és el la forma de processar-se. La forma de processar-se pot ser a través de CPU, GPU o bé amb SAH-BVH-GPU.

En aquesta primera iteració només s'implementa una part del Cas d'Ús, que és ressaltada en negreta al flux de la Taula 3. Aquesta part no té en compte les diferents formes de processar el render. Així doncs, esdevé una petita part del desenvolupament comuna i necessària en totes les iteracions posteriors.

Taula 3. Cas d'Ús general: Primera part ressaltada

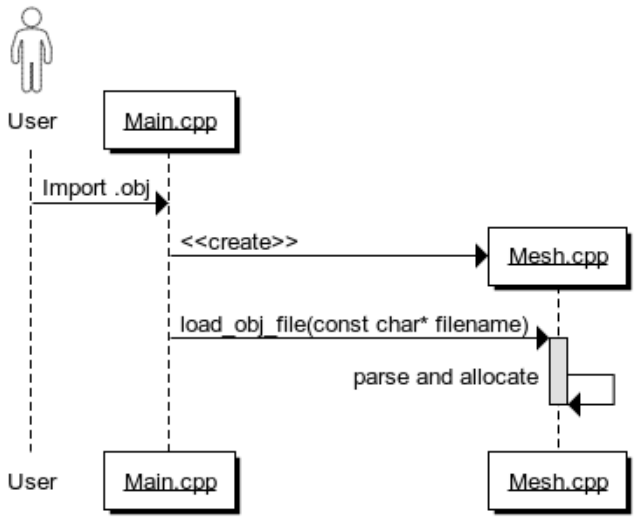
CAS D'ÚS #1	**Renderitzar amb CPU**
Descripció:	Renderitza una imatge a partir d'una malla utilitzant Ray Tracing amb CPU.
Actor/s	1. Usuari, 2. Aplicació
Pre-condició	L'Usuari s'ha descarregat un arxiu *.obj* vàlid i coneix la seva ruta local. L'Aplicació s'ha compilat amb la ruta en qüestió.
Post-condició	L'Usuari pot visualitzar la imatge *.bmp* resultant.
Flux	**1. Usuari executa l'Aplicació.** **2. Aplicació llegeix *.obj*.** 3. Aplicació processa amb la CPU i efectua renderització a través de l'algorisme Ray Tracing. 4. Aplicació codifica un *bitmap*. 5. Aplicació retorna imatge resultant.

Font: Elaboració pròpia

6.2.2.2 Disseny

Es defineix el diagrama de seqüència fins arribar al segon pas del flux de la Taula 3.

Figura 15. Diagrama de seqüència per importar una malla



Font: Elaboració pròpia

El procediment per importar un *.obj* consisteix en especificar la seva ruta en local. A continuació, el programa emmagatzema una instància de la malla en temps d'execució. Per ara, no es fa cap tipus de senyal retorn de cara l'Usuari. Només es produeix la terminació del programa i s'assumeix que la lectura s'ha fet correctament.

6.2.2.3 Implementació

6.2.2.3.1 Obtenció de recursos

Molts estudis de recerca no tenen els recursos per generar o escanejar models que contenen informació 3D. Sovint, com a convenció i estàndard s'utilitza el repositori de models de *Stanford* [21]. Aquest repositori conté models que daten del 1994, i s'ha utilitzat en nombrosos casos d'estudi acadèmics o professionals per fer demostracions tècniques multidisciplinàries.

Per les raons exposades, es segueixen les línies convencionals com a font de recursos. Finalment s'utilitza una versió del *Stanford Bunny* de 144,046 triangles per servir les darreres etapes del treball.

6.2.2.3.2 Lectura de recursos

Un arxiu *.obj* està codificat en format de text, el qual possibilita la lectura directe del seu contingut. Per facilitar la lectura d'arxius, s'utilitza la llibreria de *inputs* i *outputs* de C++ *std::ifstream.*

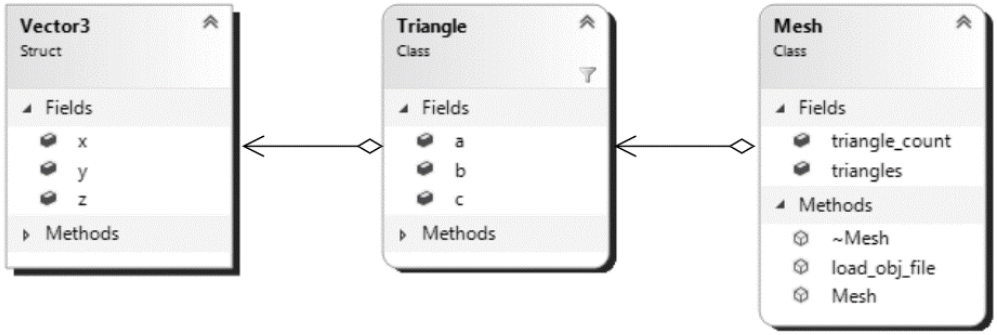
Un arxiu *.obj* pot contenir informació de geometria, coordenades UV, corbes, superfícies i altre informació que ens és irrellevant pel cas d'ús. Per interpretar la informació del arxiu, la lectura es focalitza en la *Vertex Data*.

La *Vertex Data* proveeix coordenades per la posició dels vèrtex utilitzant el prefix *v*. Per saber com s'uneixen els vèrtex i formen un complex geomètric més elevat, es fa lectura de les cares amb el prefix *f*. La codificació de les cares està composta de N referències en aquells vèrtex que estan units per una aresta.

Per emmagatzemar la informació de l'objecte en una estructura sòlida de classes reutilitzables, es generen tres nous objectes: *Vector3*, *Triangle* i *Mesh*. Aquests objectes es relacionen a través

d'agregació. Aquesta relació permet definir nivells de complexitat més elevats, essent *Mesh* el major exponent.

Figura 16. Diagrama de classes per emmagatzemar una malla triangularitzada



Font: Elaboració pròpia

6.2.2.4 Valoració

La valoració del primer cicle és positiva en tant que s'aplica estrictament la metodologia en espiral proposada, i el temps de la iteració supera l'esperat. En segon lloc, s'ha analitzat els possibles riscs d'utilitzar un sistema complex d'associacions de classes.

L'assoliment dels objectius és comprovable en les posteriors etapes, amb una representació visual del *Stanford Bunny*. En aquest punt, la metodologia en espiral autoritza planejar la següent iteració.

6.2.3 Segona iteració

6.2.3.1 Especificació

La segona iteració consta d'un objectiu molt més elevat que la primera iteració. L'objectiu consisteix en implementar un *Ray Tracer* bàsic que utilitzi la CPU per generar una imatge (ID: 3). La imatge de sortida ha de representar el model 3D que s'ha escollit a la primera iteració. Un altre requeriment funcional associat en aquesta iteració, és el camp de visió ajustable de la càmera (ID: 2).

La iteració es centra en la segona part del flux del Cas d'Ús general, que va del tercer punt al cinquè del flux.

Taula 4. Cas d'Ús general: Segona part ressaltada

CAS D'ÚS #1	**Renderitzar amb CPU**
Descripció:	Renderitza una imatge a partir d'una malla utilitzant Ray Tracing amb CPU.
Actor/s	1. Usuari, 2. Aplicació
Pre-condició	L'Usuari s'ha descarregat un arxiu *.obj* vàlid i coneix la seva ruta local. L'Aplicació s'ha compilat amb la ruta en qüestió.
Post-condició	L'Usuari pot visualitzar la imatge *.bmp* resultant.
Flux	1. Usuari executa l'Aplicació. 2. Aplicació llegeix *.obj*. **3. Aplicació processa amb la CPU i efectua renderització a través de l'algorisme Ray Tracing.** **4. Aplicació codifica un *bitmap*.** **5. Aplicació retorna imatge resultant.**

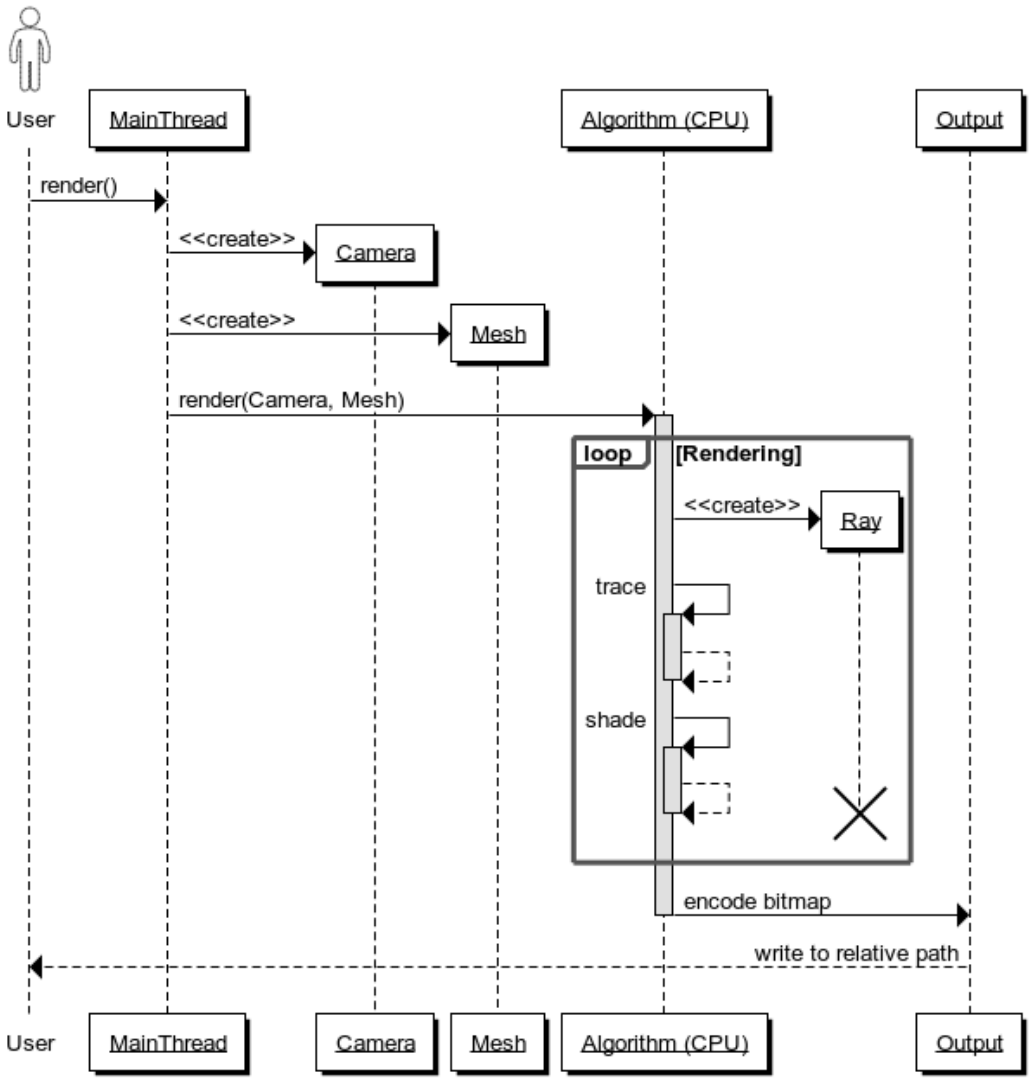
Font: Elaboració pròpia

6.2.3.2 Disseny

Seguint les pautes del Cas d'Ús, es modela un diagrama de seqüència que desglossa el funcionament intern del programa. Conèixer el funcionament intern del programa és útil per afrontar les següents etapes de implementació.

La interacció de l'Usuari amb el programa és mínima, només ha d'interactuar per executar-lo. El programa s'encarrega de crear instàncies dels recursos necessaris. A continuació, l'algorisme efectua el bucle de renderitzat. Per últim, es codifica un *bitmap* amb els resultats del render. L'Usuari pot visualitzar amb un obridor d'imatges el fitxer resultant.

Figura 17. Diagrama de seqüència del Ray Tracer amb CPU



Font: Elaboració pròpia

6.2.3.3 Implementació

Abans d'implementar codi, s'elabora un diagrama de classes del *Ray Tracer*. Aquest diagrama facilita fer un recompte de totes les utilitats necessàries per assolir l'objectiu, veure Figura 18. Fer un diagrama de classes serveix per estructurar objectes, i afronta problemes de forma *Top-Down*.

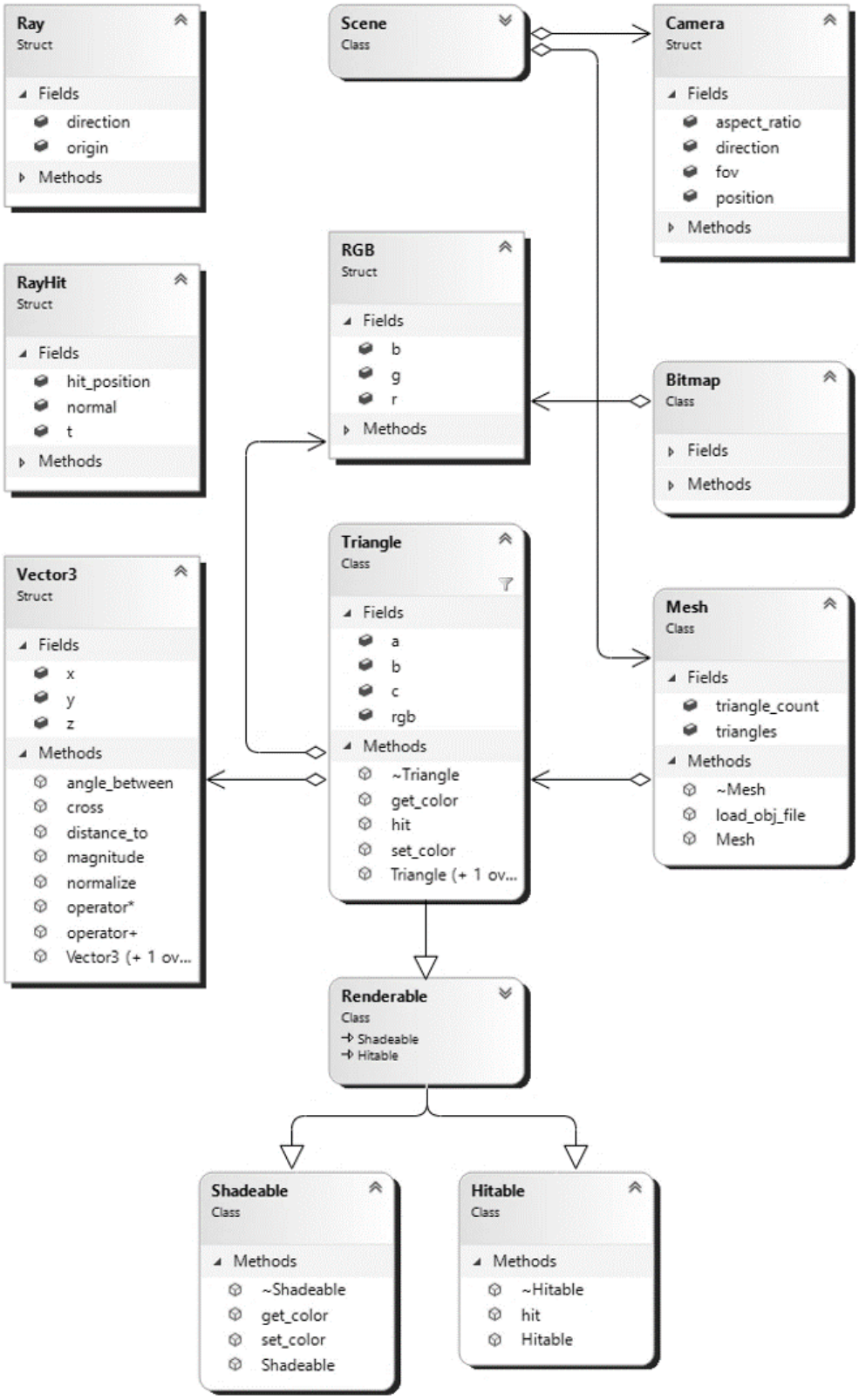
Top-Down és un estil per afrontar problemes on el producte parteix d'una descripció d'alt nivell sobre què es pretén aconseguir. Es procedeix a trencar les especificacions en trossos més petits fins que el nivell es correspon amb el vocabulari i tipus primitius del llenguatge de programació.

Un cop feta la valoració de tot allò que es vol implementar per assolir l'objectiu, es comença programant aquelles parts que són comprovables individualment. Concretament el primer que s'aborda és la implementació dels operadors matemàtics entre vectors. S'ha pres aquesta decisió perquè els vectors seran crucials per tot el desenvolupament posterior.

Fins ara, els vectors no tenien ni mètodes ni operadors, ja que no eren un requeriment de la primera iteració.

A continuació, es programa el nucli de l'algorisme de *Ray Tracing*. S'implementa expressament la forma naïf del algorisme, explicada a la secció de teoria. És aixís perquè es vol comprovar el cost potencial d'un *Ray Tracer*, i també es vol mesurar l'impacte que tenen les futures optimitzacions.

Figura 18. Diagrama de classes per el Ray Tracer amb CPU



Font: Elaboració pròpia

6.2.3.3.1 Representació de la càmera

La càmera d'un *Ray Tracer* és un objecte cèntric de l'algorisme. És fonamental per definir com es llença cada raig. Per conseqüent, la imatge de sortida serà una representació del que veu la càmera.

Els elements crucials per definir el *frustum* d'una càmera són el *Field of View*, una posició, una direcció i una resolució. És possible que alguns elements com el *Field of View* puguin ser donats d'altres formes, ja que geomètricament es descriu una equació que representa una piràmide quadrangular.

Per definir la direcció d'un raig, es necessita convertir l'espai de rasterització en espai global del món. Hi ha diverses formes de fer aquesta conversió utilitzant transformacions lineals. Tanmateix s'opta per una camí més curt gràcies les següents equacions trigonomètriques:

$$d_x = x - \frac{w}{2}$$

$$d_y = \frac{h}{2} - y$$

$$d_z = -\frac{\left(\frac{h}{2}\right)}{tan\left(fov\ \frac{1}{2}\right)}$$

Que podem reescriure:

$$d = xu + y(-v) + w'$$

$$w' = \left(-\frac{w}{2}\right)u + \left(\frac{h}{2}\right)v - \left(\frac{\left(\frac{h}{2}\right)}{tan\left(fov\ \frac{1}{2}\right)}\right)$$

On d és la direcció del Raig.

On x, y són les coordenades dels píxels.

On h, w són les dimensions determinades per la resolució de la càmera.

6.2.3.3.2 Sortida d'imatge

La codificació d'un arxiu d'imatge és un tema extens *per se*. En aquest treball es dona per suposat que la codificació d'una imatge no és objecte d'estudi.

S'opta per utilitzar un recurs provinent de tercers, que codifica un *bitmap* a partir de valors RGB. La font del qual pot ser recuperada de github.com/izanbf1803/EasyBMP sota llicència MIT.

Tal com la llicència autoritza, s'ha modificat, estès i refactoritzat el codi per suplir algunes necessitats tècniques que el recurs no proveïa inicialment.

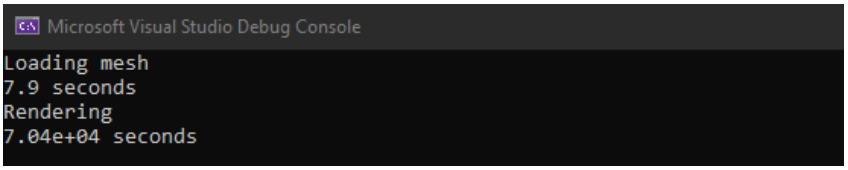
Finalment, el resultat visual obtingut de la iteració és el següent:

Figura 19. Sortida d'imatge renderitzada amb CPU

Font: Elaboració pròpia

Sortida per consola amb els detalls de temps:

Figura 20. Sortida per consola amb CPU



Font: Elaboració pròpia

El model *Stanford Bunny* que es veu a la Figura 19 conté 144,046 triangles, i la imatge té una resolució de 512 per 512 píxels. Amb aquesta configuració li hem demanat a la CPU de realitzar 40 mil milions de còmputs d'interseccions en sèrie.

Com podem veure, el temps total de renderitzar és molt elevat. Ha trigat un total de 20 hores i 55 minuts en acabar el procés. Vegem a continuació una taula amb la mateixa informació que en futures etapes servirà per analitzar i comparar els resultats.

Taula 5. Resultats obtinguts amb CPU

Model	**Triangles**	**Resolució**	**Mètode**	**Temps**
Stanford Bunny	144,046	512 x 512	CPU	20h 55m

Font: Elaboració pròpia

6.2.3.4 Valoració

En aquesta iteració s'ha pogut asserir que la implementació naïf del *Ray Tracing* és inviable a temps real. Amb el matís d'estar sotmès a un gran nombre de triangles. Aquest experiment concorda amb la teoria exposada, i reafirma el cost potencial d'un *Ray Tracer*.

El primer que cal remarcar sobre la segona iteració és l'enorme diferència de temps de treball respecte la primera iteració. La diferència de temps es deu a que la tasca és molt més voluminosa i requereix més coneixement que l'anterior.

Com a possible millora es pot planificar una iteració en subtasques més concretes. On el completament de totes les subtasques té com a finalitat generar la imatge de sortida.

Una altre possible millora respecte la implementació, hagués sigut utilitzar la llibreria oberta de matemàtiques GLM, per comptes de programar els operadors entre vectors des de zero. De totes maneres, ens hem cenyit als requeriments principals, que demanaven el mínim ús de llibreries de tercers.

L'aspecte positiu de la iteració és que s'ha complert l'objectiu proposat. Alhora, el temps de demora es compensa amb el temps guanyat de la primera iteració. En aquest punt de de la producció, estem cenyits al calendari que s'ha previst a la planificació.

Tal com es preveu la metodologia podem passar a la següent iteració, i deixar aquesta per finalitzada.

6.2.4 Tercera iteració

6.2.4.1 Especificació

La tercera iteració consisteix en aplicar una nova forma de processament del algorisme (ID: 4). La nova forma de processament es basa en el paral·lelisme que proveeix la GPU. Es modifica la iteració anterior que utilitza la CPU fent els mínims canvis necessaris. Finalment es mesuren els resultats obtinguts a nivell d'eficiència.

En el Cas d'Ús que es mostra a continuació, es ressalta en negreta la part del flux que s'implementa en la iteració.

Taula 6. Cas d'Ús amb GPU

CAS D'ÚS #2	**Renderitzar amb GPU**
Descripció:	Renderitza una imatge a partir d'una malla utilitzant Ray Tracing amb GPU.
Actor/s	1. Usuari, 2. Aplicació
Pre-condició	L'Usuari s'ha descarregat un arxiu *.obj* vàlid i coneix la seva ruta local. L'Aplicació s'ha compilat amb la ruta en qüestió.
Post-condició	L'Usuari pot visualitzar la imatge *.bmp* resultant.
Flux	1. Usuari executa l'Aplicació. 2. Aplicació llegeix *.obj*. **3. Aplicació processa amb la GPU i efectua renderització a través de l'algorisme Ray Tracing.** 4. Aplicació codifica un *bitmap*. 5. Aplicació retorna imatge resultant.

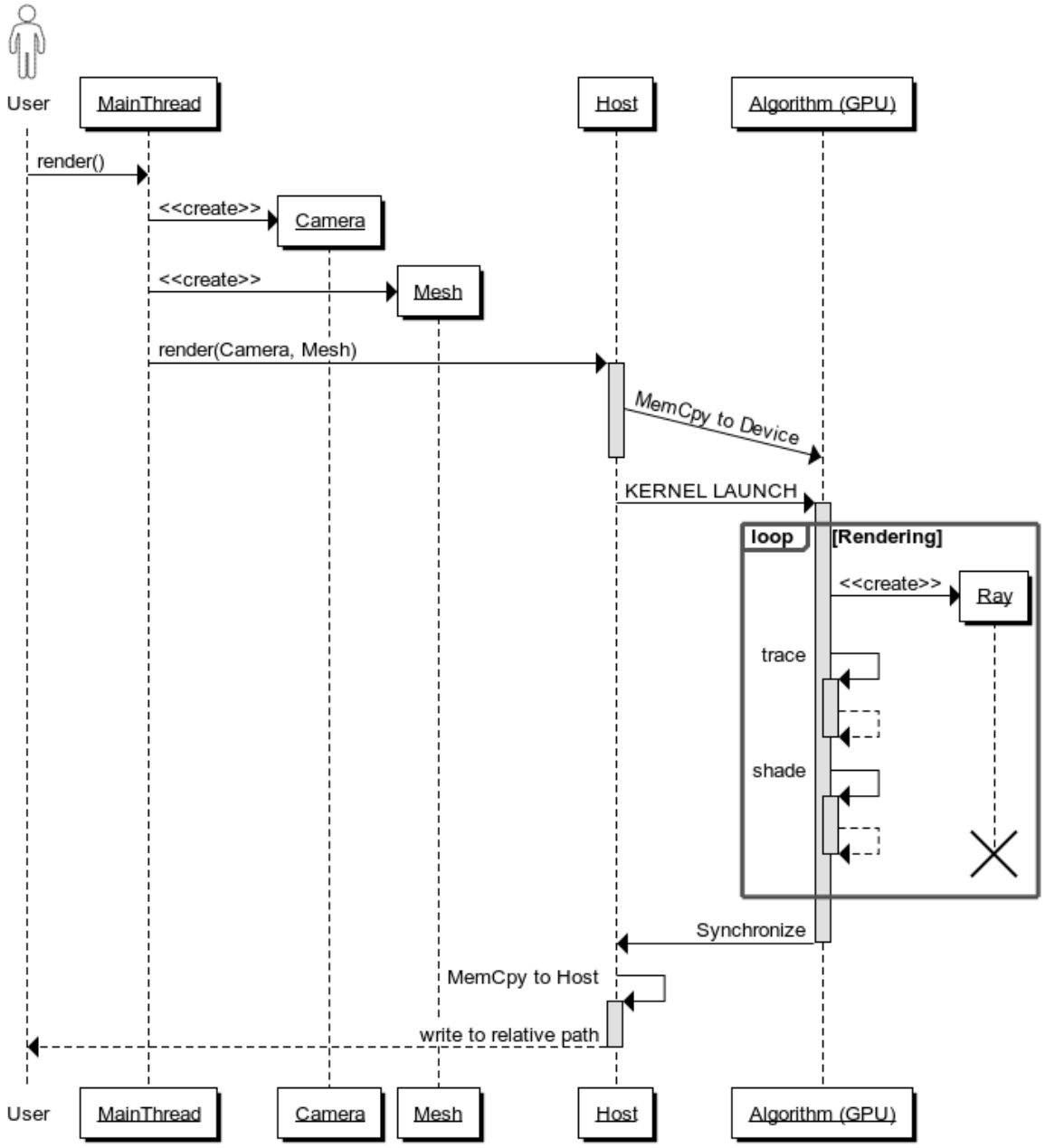
Font: Elaboració pròpia

De cara l'Usuari la interacció amb l'aplicació es manté igual, mentre que de cara l'Aplicació canvia la forma de processament de l'algorisme, ara amb GPU.

6.2.4.2 Disseny

Malgrat que el Cas d'Ús és similar a la iteració anterior, la implementació del programa es modifica considerablement. Es dissenya el funcionament intern del programa a baix nivell per entendre com serà la seva implementació.

Figura 21. Diagrama de seqüència del Ray Tracer amb GPU



Font: Elaboració pròpia

En el diagrama s'introdueixen dues fases essencials pel funcionament de l'algorisme a la GPU. En elles s'assigna i es copia la memòria des del *Host* al *Device* perquè la GPU hi tingui accés a la seva memòria. Finalment, es sincronitza l'execució dels *threads* i es copia la memòria generada pel *Device* al *Host*. La informació generada composa la imatge formada per píxels, que posteriorment es codifica i es guardarà en local en un *bitmap*.

6.2.4.3 Implementació

6.2.4.3.1 Preparació de l'entorn CUDA

Abans de començar la implementació, és indispensable preparar l'entorn de desenvolupament CUDA i assegurar la seva compatibilitat.

Per assegurar la compatibilitat, s'ha d'escollir una versió suportada per la GPU i s'ha de fer una instal·lació neta seguint les guies de la documentació oficial. Per últim, verifica la instal·lació compilant projectes proporcionats per la pròpia guia.

Un cop feta la comprovació de compatibilitat, es procedeix fent una sèrie de consultes a través de CUDA sobre el *Device*. Les consultes proporcionen informació referent a la GPU i les seves capacitats computacionals. La informació serveix per optimitzar el codi i assumir els recursos dels quals disposa la targeta gràfica. També serveix per assegurar que la GPU encaixa amb el nostre hardware.

Figura 22. Consulta de Device

Font: Elaboració pròpia

A la Figura 22 es mostren algunes capacitats que disposa la GPU utilitzada en aquest treball. Primerament es mostra el número de *Devices* el qual indica que només hi ha una sola GPU, però poden ser múltiples. En segon lloc, el model de GPU de *Nvidia*. També es mostra la velocitat d'execució i l'amplada del bus. Per últim es mostra el número màxim de *threads* per cada *block*. Una altre característica que no surt a la figura, és la *compute capability* de 5.0, que

descriu els límits físics de la GPU. Els límits físics són indicats pel número de *Threads* per *Warp* o bé el número màxim de *threads* per *Multiprocessor* entre d'altres característiques del hardware. A continuació es mostra la taula completa de límits imposats per la *compute capability* 5.0:

Taula 7. Límits de la Compute Capability

Physical Limits for GPU Compute Capability:	***5.0***
Threads per Warp	*32*
Max Warps per Multiprocessor	*64*
Max Thread Blocks per Multiprocessor	*32*
Max Threads per Multiprocessor	*2048*
Maximum Thread Block Size	*1024*
Registers per Multiprocessor	*65536*
Max Registers per Thread Block	*65536*
Max Registers per Thread	*255*
Shared Memory per Multiprocessor (bytes)	*65536*
Max Shared Memory per Block	*49152*
Register allocation unit size	*256*
Register allocation granularity	*warp*
Shared Memory allocation unit size	*256*
Warp allocation granularity	*4*

Font: Obtinguda de CUDA Occupancy Calculator

6.2.4.3.2 Gestió de memòria

Tal com s'explica en el diagrama de seqüència, hi ha un pas previ a la computació de l'algorisme. En aquest pas s'assigna i es còpia la memòria de la malla i la càmera a la GPU. També es reserva espai de memòria per dipositar la imatge resultant. De no ser així, no podríem extraure els resultats del *bitmap*.

Quan s'accedeix a *arrays* de dues o tres dimensions a CUDA, les transaccions de memòria són més ràpides si les files estan alineades. CUDA ofereix una forma de calcular el *Pitch* i el *Padding* necessari per alinear les files i columnes en adreces de memòria. Les dimensions poden ser de 64 fins a 512 bytes depenent de les restriccions del hardware.

Com a conseqüència, es desaprofita espai de memòria i es complica l'accés de cara al programador. Com a avantatge la l'accés a memòria és més eficient per CUDA.

Al nostre cas concret, per reservar espai de memòria de la imatge de sortida s'emmagatzemen píxels de tres bytes. S'utilitzen 8 bits (*uint8_t*) per cada un dels tres paràmetres RGB.

Taula 8. Memòria per la imatge de sortida

Element	**Quantitat**
sizeof(RGB)	3 bytes
IMG_WIDTH	512 px
IMG_HEIGHT	512 px
Memòria total necessària	786432 bytes

Font: Elaboració pròpia

Es considera un cas especial degut a que les dimensions de la imatge són múltiple de 64. Significa que les dimensions són adequades per estar alineades a la memòria. El pitch resultant és de 3072 bytes i el *Padding* és de zero bytes.

Una fórmula per calcular el percentatge d'espai desaprofitat que s'ha utilitzat és la següent:

$$(r\%b)/r = wPct$$

On b (*boundary* en anglès) és el límit de les adreces de memòria.

On r (*row* en anglès) és la mida de les files.

On $wPct$ (*waste* en anglès) és el percentatge desaprofitat de memòria total.

Per accedir a *pitched memory* s'utilitza la següent expressió:

$$T^*\ e = (T^*)\big((char^*)addr + r \cdot pitch\big) + c;$$

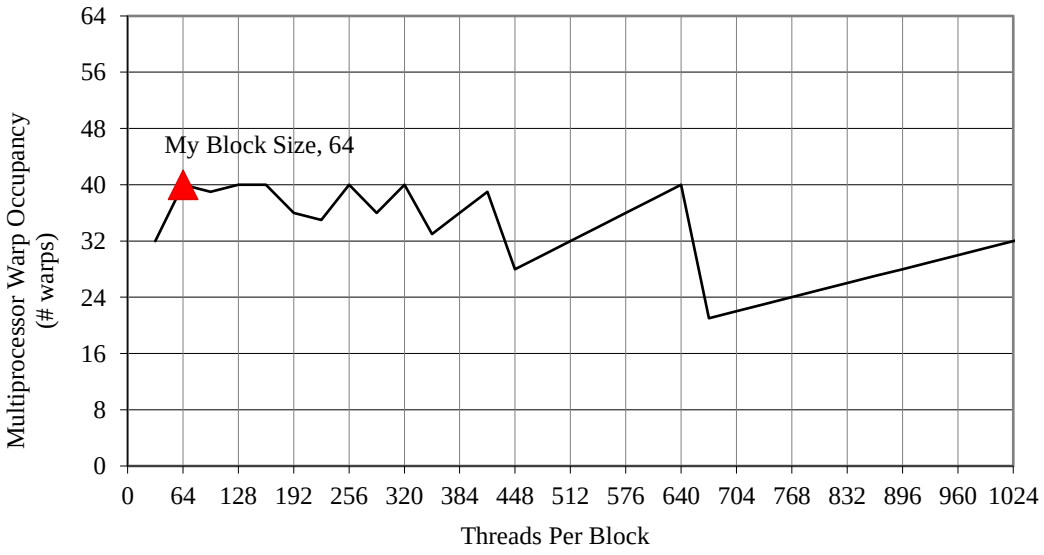
Donada una columna c i una fila r d'una *array* composta d'elements e de tipus T es pot obtenir l'adreça de l'element desitjat. Aquesta expressió s'utilitza a la implementació del *kernel* de CUDA per generar el nostre *bitmap*.

Un altre aspecte que es té en compte és la minimització de l'ús de registres en el *kernel*. Un *Streaming Multiprocessor* té 65536 registres a disposició dels *Threads*. No és trivial conèixer el número de registres que pot arribar a requerir un *Thread*. Per saber-ho en exactitud és convenient utilitzar eines de *Nvidia* com ara la compilació de NVCC amb la *flag: –ptxas-options=-v.*

Optimitzar la utilització dels registres requereix coneixement a baix nivell del funcionament de CUDA. El codi que genera NVCC per facilitar la depuració a baix nivell és SASS o PTX.

També és útil conèixer la ocupació de memòria d'un *kernel* en base a les característiques del hardware. *Nvidia* proveeix eines com *CUDA Occupancy Calculator* per saber amb detall l'impacte que té la gestió de la memòria. El llançament del nostre *kernel* obté una ocupació del 63%, veure Figura 23.

Figura 23. Impacte de la mida de Block variant



Font: CUDA Occupancy Calculator

El pic on es situa el nostre cas està marcat amb un triangle vermell. La resta de punts de la gràfica representen un rang de possibles variacions.

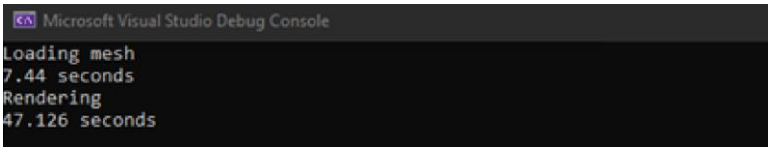
El percentatge és el resultat del nombre de *warps* actius (40) sobre el màxim nombre de *warps* per *Streaming Multiprocessor* (64). La causa de la limitació resideix en que la utilització de registres per *thread* és 42. Per tal d'obtenir una ocupació teòrica del 100% de la GPU, s'ha de reduir el nombre de registres per *thread* a 32. D'aquesta forma tindrem tots els *stream processsors* treballant simultàniament. Malauradament, la *compute capability* ens limita pel màxim nombre de registres per *Streaming Multiprocessor*.

La ocupació no necessàriament significa major rendiment. No és aconsellable buscar la màxima ocupació en tots els casos, ja que l'ús eficient dels recursos pot dependre del projecte. En el nostre cas s'ha pogut observar una diferència notable de rendiment al provar amb altres configuracions del *kernel*.

6.2.4.3.3 Sortida d'imatge

La sortida d'imatge és idèntica a la Figura 19. La velocitat amb GPU millora considerablement respecte la CPU.

Figura 24. Sortida per consola amb GPU



Font: Elaboració pròpia

Anotem en una taula la informació rellevant que en futures etapes servirà per analitzar i comparar els resultats.

Taula 9. Resultats obtinguts amb GPU

Model	**Triangles**	**Resolució**	**Mètode**	**Temps**
Stanford Bunny	144,046	512 x 512	GPU	47s

Font: Elaboració pròpia

6.2.4.4 Valoració

En retrospectiva, aquesta iteració ha suposat un gran esforç per conèixer el desenvolupament amb dos compiladors diferents, dos processadors, i la gestió de memòria en una arquitectura desconeguda fins aleshores.

La valoració de la tercera iteració ha estat exitosa en tant que s'han assolit els objectius proposats. A pesar d'haver arribat als objectius, ens hem trobat amb reptes pel camí que han portat més temps del esperat en resoldre's.

6.2.4.4.1 Reptes i desviacions

El primer repte que ha suposat un extra de temps és el desconeixement tècnic de CUDA sobre els objectes o mètodes virtuals. Per superar el repte s'ha tingut de llegir amb deteniment alguna part de la extensa de la documentació oficial. Un cop s'ha comprès aquell aspecte de la documentació hem pogut seguir amb la implementació.

També ha sigut un repte analitzar l'eficiència del *kernel* i depurar el codi a la GPU. Per depurar codi en un processador extern s'ha de configurar l'IDE *Visual Studio* amb extensions de *Nvidia*. La configuració ha estat lluny de ser trivial, s'ha necessitat molta recerca per activar les opcions necessàries perquè funcioni tot correctament.

Un cop s'ha fet la implementació, l'anàlisi d'ocupació i eficiència ha demostrat que encara es pot optimitzar el codi en gran mesura. Malauradament, no s'ha aconseguit reduir el nombre de registres en els *threads* de la GPU per falta de temps.

Un altre repte que ha complicat el desenvolupament, és un error imprevist provocat per el TDR de Windows (*Timeout Detection and Recovery*). Aquest error es deu a que s'excedeix el temps de càlcul en un *thread*. Windows interpreta que el procés s'ha congelat i talla abruptament l'execució sense indicar els motius. Les conseqüències van més enllà del tall del procés, en algunes ocasions s'ha perdut la senyal de les pantalles produint artefactes visuals. Després de fer investigació i entès l'origen del problema, s'ha trobat una solució modificant el registre de Windows.

6.2.5 Quarta iteració

6.2.5.1 Especificació

La quarta iteració consisteix en implementar l'última gran optimització a l'algorisme (ID: 5). S'elabora un arbre per subdividir l'espai i efectuar consultes de rajos més eficients. S'aprofita l'algorisme existent amb el potencial de la GPU combinant el potencial del BVH. S'espera que millori la velocitat de càlcul al final de la iteració.

La Taula 10 ressalta en negreta el que s'implementa per al cas d'ús que es vol satisfer.

Taula 10. Cas d'Ús amb GPU i SAH-BVH

CAS D'ÚS #3	**Renderitzar amb GPU i SAH-BVH**
Descripció:	Renderitza una imatge a partir d'una malla utilitzant Ray Tracing amb GPU i SAH-BVH.
Actor/s	1. Usuari, 2. Aplicació
Pre-condició	L'Usuari s'ha descarregat un arxiu *.obj* vàlid i coneix la seva ruta local. L'Aplicació s'ha compilat amb la ruta en qüestió.
Post-condició	L'Usuari pot visualitzar la imatge *.bmp* resultant.
Flux	1. Usuari executa l'Aplicació. 2. Aplicació llegeix *.obj*. **3. Aplicació genera un SAH-BVH** **4. Aplicació processa amb la GPU i el SAH-BVH i efectua renderització a través de l'algorisme Ray Tracing.** 5. Aplicació codifica un *bitmap*. 6. Aplicació retorna imatge resultant.

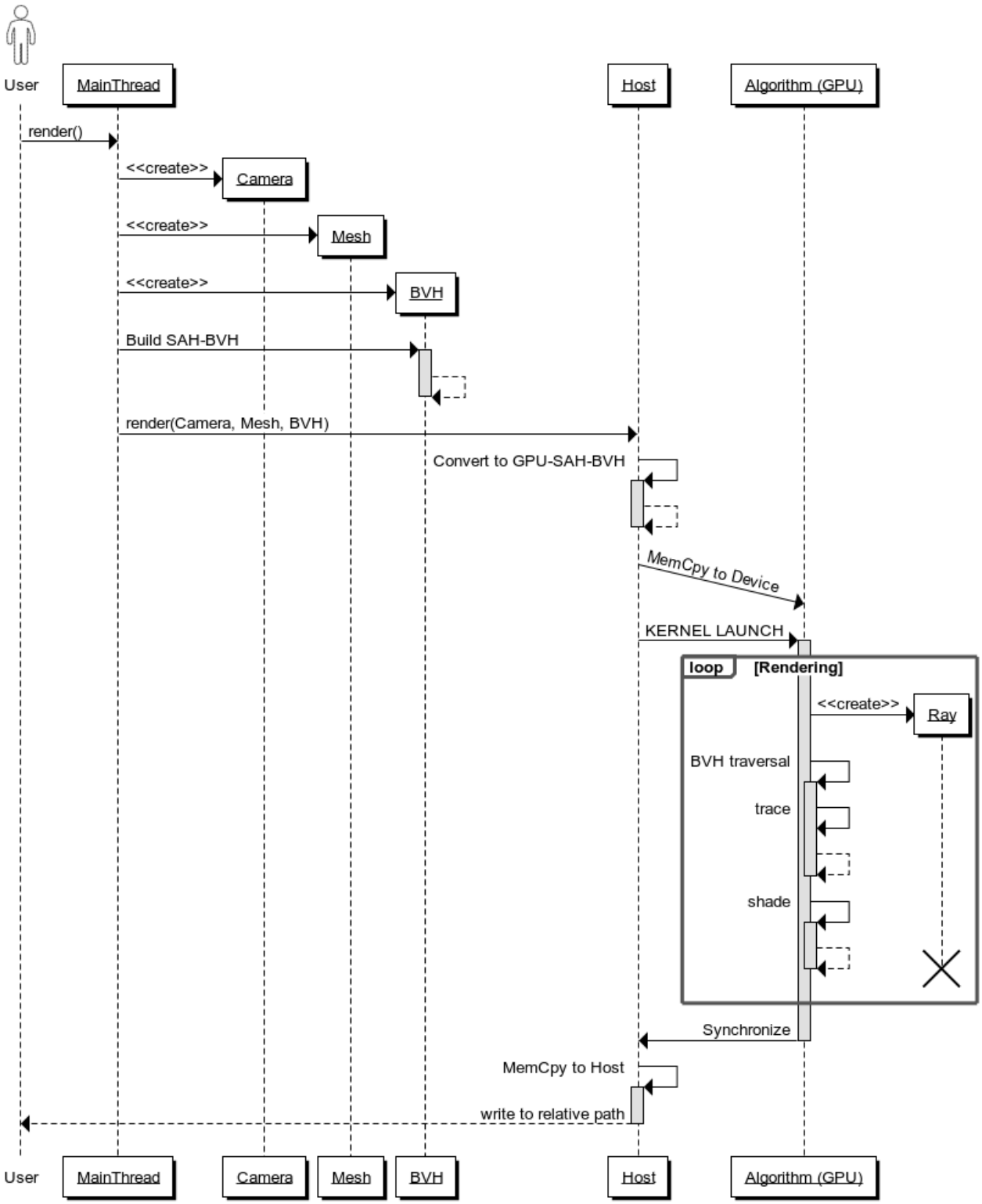
Font: Elaboració pròpia

6.2.5.2 Disseny

El disseny es veu afectat respecte les iteracions anteriors, però generalment manté la mateixa estructura. Els passos nous que s'introdueixen són la construcció del SAH-BVH, i l'adaptació

perquè sigui compatible amb la GPU de forma eficient. També s'introdueix la necessitat de fer un recorregut per l'arbre a dins de la GPU per calcular les interseccions amb els raigs.

Figura 25. Diagrama de seqüència del Ray Tracer amb GPU i SAH-BVH



Font: Elaboració pròpia

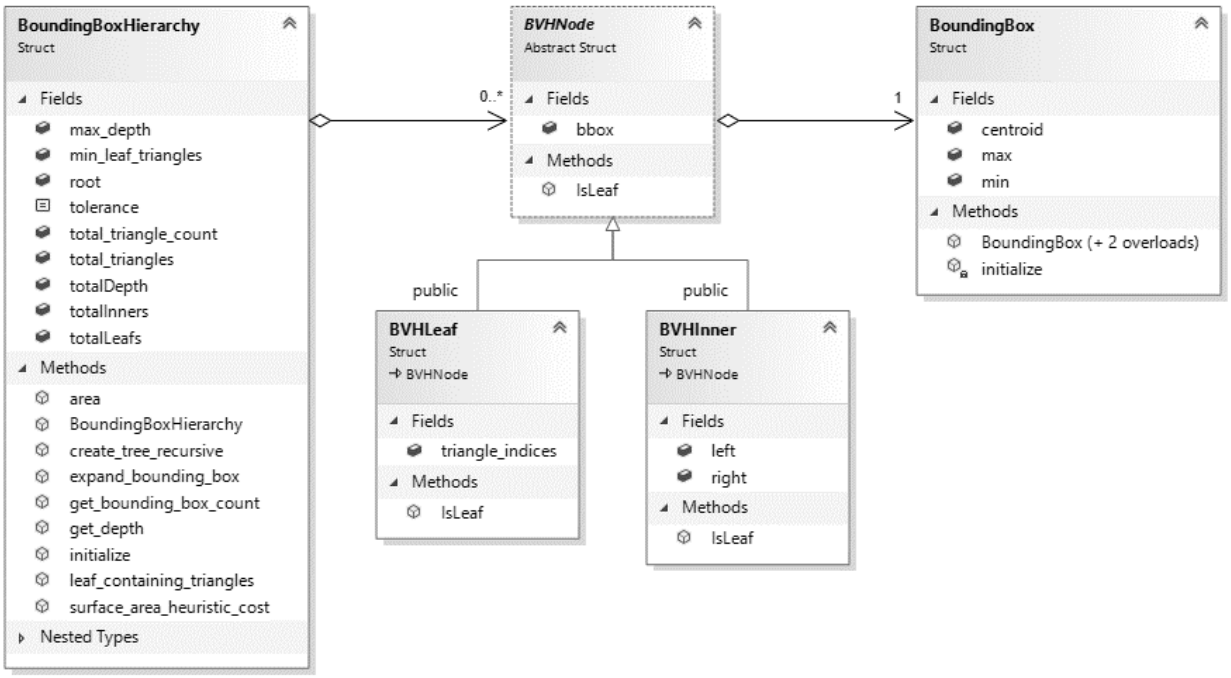
6.2.5.3 Implementació

Per afrontar una implementació d'aquesta magnitud, es divideix la feina en tres etapes. La primera etapa és la creació del SAH-BVH a la CPU. Això facilita la depuració abans de moure'ns cap a la GPU. La segona etapa és l'adaptació del BVH a la GPU. Per últim s'implementa el recorregut del BVH a la GPU.

6.2.5.3.1 Construcció de SAH-BVH a la CPU

Prèviament a la implementació del BVH, s'elabora un diagrama de classes. El diagrama modela els nous objectes que participen a la creació de la nova estructura de dades. A major escala, es troba la *BoundingBoxHierarchy* formada per un conjunt de nodes. Els nodes poden ser fulla o nodes interns de l'arbre. En funció de l'especialització, el node pot contenir una informació o bé una altre. Un node sense especialització no és res, per tant s'hereta d'una classe abstracte. Sabem que tot node té una *BoundingBox,* i per suplir la necessitat fem una agregació amb la classe base.

Figura 26. Diagrama de classes del SAH-BVH per CPU



Font: Elaboració pròpia

La classe *BoundingBoxHierarchy* es pot configurar amb paràmetres com la profunditat màxima de l'arbre així com el mínim nombre de triangles per esdevenir una fulla. Per qüestions de precisió numèrica també es disposa d'una tolerància per calcular el *Surface Area Heuristic*.

A continuació s'explica el procediment de construcció del SAH-BVH. Per construir el BVH es crea una instància del BVH amb els paràmetres desitjats. El constructor no té cap més responsabilitat més enllà de guardar la configuració. La construcció del BVH comença amb el mètode *BoundingBoxHierarchy::Initialize()*. El mètode *Initialize* encomana la tasca a un mètode auxiliar per efectuar la construcció de l'arbre de forma recursiva. La recursivitat construeix l'arbre en ordre de profunditat.

Per optimitzar la intersecció raig-triangle es minimitza el cost a través de l'heurístic SAH. L'heurístic ajuda a determinar la millor repartició de triangles entre els fills dret i esquerra. Per determinar el tall i dividir una *BoundingBox* s'avaluen tots els possibles talls perpendiculars als eixos. D'aquesta manera es garanteix trobar el tall més òptim fent totes les comprovacions possibles.

Per efectuar talls tenint en compte el nivell de l'arbre i la precisió, s'utilitza la següent fórmula:

$$\left(T_f - T_i\right) / C / (P + 1.0) > \varepsilon$$

On el rang de talls $\in [T_i, T_f]$ ve determinat per les dimensions de la *BoundingBox*.

On la constant C s'escull per delimitar la mida de la graella.

On ε és la tolerància que ha de superar el tall per ser vàlid.

On P és la profunditat en el nivell de recursivitat que es troba.

Per calcular el SAH s'apliquen altres fórmules descrites al marc teòric amb més detall.

Finalment, s'obté una jerarquia de punters que emmagatzemen nodes interns o fulles. El BVH s'ha construït en base a alguns principis i bones pràctiques per assolir codi net, amb l'objectiu de facilitar la feina de cara al programador. Malauradament, a la GPU no sempre es pot escriure el codi ideal, ja que es prioritza l'eficiència per damunt de tot. A continuació s'explica com s'afronta aquest repte.

6.2.5.3.2 Adaptació de BVH a la GPU

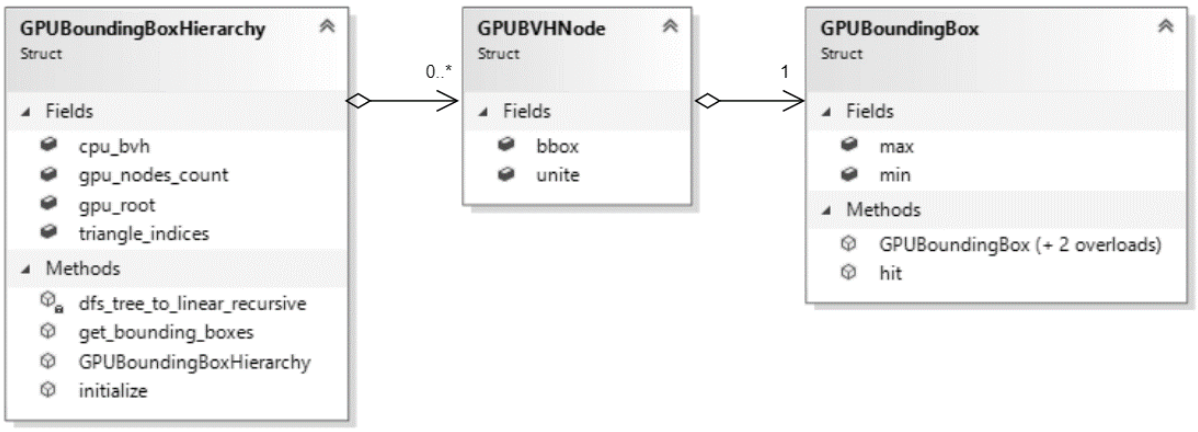
Una de les parts més complicades de la acceleració del traçat de rajos és com emmagatzemar el BVH a la GPU. El format ha de trobar un balanç entre maniobrabilitat i eficiència. Com hem

vist, a la CPU el BVH es guarda en forma de jerarquia de punters començant per l'arrel. Els nodes referencien a altres nodes fins a referenciar els triangles de les fulles. Aquesta estructura portada a la GPU pot provocar un estat de segmentació de memòria indesitjat.

A la GPU es genera una estructura de dades alineada i indexada de forma coherent com són els *Arrays*. La implementació d'aquest treball aplana la profunditat de l'arbre en una estructura unidimensional de nodes. Per mantenir l'estructura arbòria inicial s'emmagatzemen els nodes en ordre *Depth First Search*.

Per definir els nodes ja no s'utilitza l'herència de classes, s'utilitza una versió optimitzada que ocupa memòria fixe amb una unió de *structs*. Tant si és de tipus fulla com si és un node intern un node ocupa 32 bytes. També s'utilitza una versió optimitzada de la *BoundingBox* amb mètodes per calcular la intersecció raig-triangle al *device.* A continuació es mostra la nova estructura dissenyada per l'adaptació a la GPU.

Figura 27. Diagrama de classes del SAH-BVH per GPU



Font: Elaboració pròpia

La classe *GPUBoundingBoxHierarchy* és la versió optimitzada per la GPU de la classe *BoundingBoxHierarchy*. Per convertir l'estructura es fa l'aplanament a través del mètode *GPUBoundingBoxHierarchy::dfs_tree_to_linear_recursive()*.

Queda pendent fer la diferenciació de la tipologia de nodes per quan fem el recorregut. Per saber si un node és una fulla es modifica un sol bit. Concretament, es modifica el primer bit de més a l'esquerra de la variable que té el número de triangles. D'aquesta manera sabrem interpretar què representen les dues variables d'enters. Assumim però, que a partir d'ara no farem ús del major número negatiu `0x80000000` en segon complement. S'utilitza l'operador *Bitwise OR* de C++ per modificar el bit.

Figura 28. Estructura de representació de nodes optimitzada per GPU

```
1   struct GPUBVHNode
2   {
3           GPUBoundingBox bbox;
4
5           union
6           {
7                   struct
8                   {
9                           unsigned int left_index;
10                          unsigned int right_index;
11                  } inner;
12
13                  struct
14                  {
15                          unsigned int count;
16                          unsigned int triangle_index;
17                  } leaf;
18
19          } unite;
20  };
```

Font: Elaboració pròpia

Finalment, la nova estructura de dades és assignada a la memòria global de la GPU. Malgrat que la memòria constant és d'accés més lent, s'ha optat fer-ho d'aquesta forma per simplificar la lectura. Una altre opció és utilitzar *Texture Memory* o *Shared Memory*. A la següent etapa s'exposa la implementació del recorregut del BVH optimitzat per GPU que acabem de veure.

6.2.5.3.3 Recorregut del BVH a la GPU

El recorregut del BVH és l'essència de l'acceleració del *Ray Tracing*. El recorregut determina si un raig fa intersecció amb un triangle, o no. Hi ha moltes formes d'afrontar aquest repte,

però cadascuna té les seves limitacions. Una forma elegant d'afrontar el recorregut és a través de la recursivitat. Malauradament, la recursivitat fa ús del *stack* de memòria i pot ser contraindicat a la GPU perquè el compilador és incapaç de preveure la màxima profunditat de recursió per reservar espai. Una pràctica comuna és limitar la profunditat de recursió per evitar un *stack overflow*.

En el nostre cas, no es fa ús de la recursivitat de CUDA. S'implementa un recorregut a base de iteracions. Es defineix un *stack* propi, la mida del qual és fixe i està definida en un *macro*. Si la profunditat supera la mida definida, es finalitza l'execució amb coneixement de l'excepció. L'*stack* és un *array* emmagatzemat al *stack* de la GPU. No és un *stack* provinent de *std::stack*.

L'algorisme del recorregut fa *push* dels nodes que han fet intersecció amb la *Bounding Box*. Mentre hi hagi nodes per visitar al *stack*, el bucle continua fins que s'exhaureix. Quan un node és visitat s'elimina del *stack*. Quan s'exhaureix *l'stack* vol dir que no queden nodes per visitar.

Quan l'algorisme es topa amb un node de tipus fulla, es procedeix a comprovar totes les interseccions amb els triangles que conté la fulla. Arribat aquest punt, si encara no s'ha trobat cap intersecció, seguim exhaurint l'*stack*. Un cop es troba una fulla es calcula la intersecció del raig amb tots els triangles continguts en ella. El nombre de triangles contingut en una fulla és molt baix gràcies al BVH i la complexitat lineal és molt baixa. Finalment, si s'ha donat una intersecció es calcula el color del píxel.

Si s'exhaureix l'*stack* i el raig no ha trobat cap intersecció amb un triangle, el píxel es pinta amb color negre. Es considera que el raig ha topat amb el cel que es troba a una distància infinita.

6.2.5.3.4 Sortida d'imatge

L'avantatge d'utilitzar un BVH és que la sortida de la imatge obté un resultat més ràpid, amb la contradicció del seu cost de construcció. La construcció del BVH proporciona l'acceleració de les consultes entre raig-triangles. A continuació es mostra la taula de temps amb GPU-SAH-BVH.

Figura 29. Sortida per consola de l'execució del GPU-SAH-BVH

Font: Elaboració pròpia

Com podem veure, l'arbre s'ha generat amb onze nivells de profunditat i ha trigat 246 segons en construir-se. Malgrat hi ha formes més òptimes per construir un BVH, no és objecte d'estudi la seva acceleració *per se*. El BVH ha permès que el temps de *Ray Tracing* sigui menor que un segon, i ha produït una resultat idèntic a la Figura 19.

Anotem els resultats en una taula seguint el procediment que hem fet al final de cada iteració.

Taula 11. Resultats obtinguts amb GPU-SAH-BVH

Model	**Triangles**	**Resolució**	**Mètode**	**Temps**
Stanford Bunny	144,046	512 x 512	GPU-SAH-BVH	0.752s

Font: Elaboració pròpia

6.2.5.4 Valoració

El primer que cal destacar és que s'ha assolit tots els objectius proposats arribats aquest punt. Satisfactòriament hem pogut comprovar les principals formes d'acceleració del *Ray Tracer*. Hem pogut comprovar l'impacte que té el paral·lelisme a la GPU en conjunt amb l'acceleració proporcionada pel BVH.

6.2.5.4.1 Reptes i desviacions

Ens hem trobat amb molts reptes i obstacles durant aquest treball que han estat superats amb els temps previstos. El tipus de reptes han estat principalment relacionats amb la recerca sobre l'objecte d'estudi, i entendre el funcionament d'un nou hardware i nous algorismes. També hi ha hagut reptes relacionats amb la depuració de codi a la GPU.

Un dels grans costos del desenvolupament de la iteració ha sigut resoldre un error a l'hora d'executar el *kernel.* El problema a simple vista era indesxifrable perquè l'execució del *kernel* es tallava abruptament sense indicacions. S'ha aconseguit resoldre a base de analitzar el codi durant tres dies. Un cop conegut l'error s'ha comprès la causa i s'ha pogut resoldre. A pesar de resoldre el problema amb èxit, es considera un problema poc trivial de detectar si es té poca experiència.

A continuació s'expliquen més detalls sobre l'error que ha provocat un petit descarrilament de temps. L'error era provocat per un *memory leak* a la GPU amb l'*stack* de memòria per recórrer el BVH. A la GPU s'estava assignant memòria al *Heap* sense eliminar memòria. La no eliminació acumula memòria a mida que s'executa la resta de *threads*. Finalment, el problema s'ha resolt emmagatzemant al *Stack* de memòria de la GPU.

Quan s'emmagatzema al *Stack,* la memòria es borra quan surt de *scope* i no és necessària la recol·lecció d'escombraries manual. Un altre aspecte a diferenciar entre el *Heap* i l'*Stack* a la GPU, és que el *Heap* resideix a la memòria global. En canvi, l'*Stack* s'assigna a la memòria local per *thread* i proporciona un accés més ràpid.

7 Resultats

En aquest apartat es fa una valoració que recull els resultats obtinguts en totes les iteracions. Posteriorment es discuteixen possibles millores i futures etapes del treball.

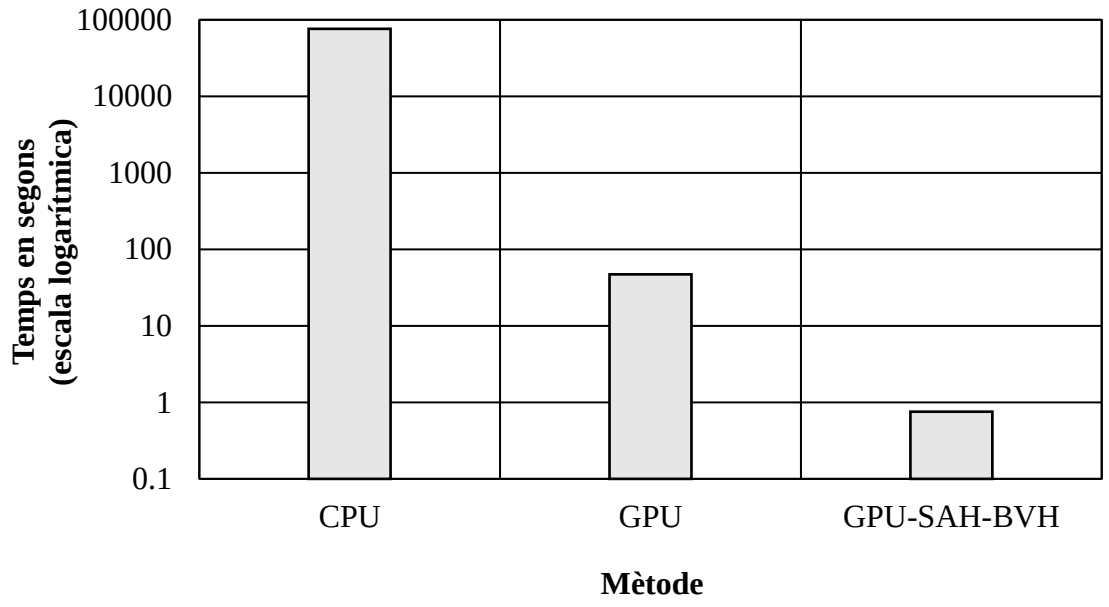
Taula 12. Resultats obtinguts definitius

Model	**Triangles**	**Resolució**	**Mètode**	**Temps**
Stanford Bunny	144,046	512 x 512	CPU (serial)	20h 55m
Stanford Bunny	144,046	512 x 512	GPU	47s
Stanford Bunny	144,046	512 x 512	GPU-SAH-BVH	0.752s

Font: Elaboració pròpia

El primer que salta a la vista és que el gràfic mostra una proporció dramàtica amb el primer mètode a través de CPU, respecte els altres dos mètodes amb GPU. La diferència es deu a que el paral·lelisme massiu de la GPU és més adient pel traçat de rajos que l'execució en sèrie.

Figura 30. Gràfic comparatiu del temps en escala logarítmica



Font: Elaboració pròpia basada en resultats obtinguts

A pesar que visualment la diferència entre els dos mètodes que usen GPU sembla imperceptible, la diferència s'ha reduït seixanta tres vegades. Només amb GPU-SAH-BVH s'obra la viabilitat al *Ray Tracing* a temps real a 1.3 FPS. Si ho comparem amb altres implementacions existents continua essent un nombre molt baix de fotogrames per segon. El temps hauria de reduir-se a 0.016 segons per arribar a 60 FPS.

Per reduir el temps a mil·lisegons s'haurien d'implementar tècniques més elaborades. Les tècniques aplicades en aquest treball no van més enllà de la implementació naïf de cada mètode respectiu. Les tècniques més elaborades que existeixen estan profundament integrades als controladors de hardware de les GPUs de última generació. És difícil sinó impossible conèixer tots els secrets al darrera de la construcció d'un BVH a l'arquitectura Turing, més enllà de la idea fonamental.

El punt on finalitza el treball obre la possibilitat d'afegir una gran varietat de funcionalitats i característiques noves. Entre elles, com s'ha mencionat a la Taula 2, es pot implementar *Path Tracing* per generar un model d'il·luminació fotorealista. Cal recordar que l'acceleració del BVH és en previsió al càlcul de més rebots secundaris després del primer impacte.

L'essència del *Ray Tracing* és la versatilitat per simular efectes de llum, i no s'ha arribat a veure en cap moment del treball.

Aquest treball només s'ha focalitzat en el llançament de rajos primaris. Existeixen mètodes més eficients per computar els rajos primaris utilitzant un model híbrid amb la rasterització. Malgrat, l'elegància del *Ray Tracing* suposa una gran avantatge per no haver de fer un cas especial pels primers rajos.

7.1 Conclusions

La conclusió global del treball és que s'ha pogut asserir l'enorme impacte d'utilitzar la GPU i el BVH, imprescindibles pel *Ray Tracing* a temps real a dia d'avui.

Malauradament, el *Ray Tracing* en la seva màxima expressió encara està lluny de ser possible. La màxima expressió del *Ray Tracing* es considera aquella que funciona a temps real, en resolució nativa i amb difusió de radiació. La qual és inviable en hardware de consum, i probablement ho serà durant més anys.

Avui en dia, l'estat de l'art del *Ray Tracing* a temps real implica moltes ajudes alternatives perquè funcioni. Les grans produccions de videojocs AAA que usen la tecnologia RTX es complementen de tècniques híbrides basades en la rasterització. Sovint només utilitzen *Ray Tracing* per resoldre algun tipus de reflexió amb efecte mirall.

El coneixement del *Ray Tracing* es remunta a segles d'antiguitat.
A pesar de ser ben conegut, no ha sigut fins l'actualitat del 2018, la primera vegada a la història que s'ha portat a nivell de producció a temps real
amb el suport de les empreses més grans del sector.

What a time to be alive!

8 Bibliografia

[1] K. Zibrek, S. Martin, i R. McDonnell, «Is Photorealism Important for Perception of Expressive Virtual Humans in Virtual Reality?», *ACM Trans. Appl. Percept.*, vol. 16, núm. 3, p. 14:1-14:19, set. 2019, doi: 10.1145/3349609.

[2] P. Shirley i S. Marschner, *Fundamentals of Computer Graphics*, 3rd ed. USA: A. K. Peters, Ltd., 2009.

[3] A. M. Noll, «Computers and the Visual Arts», *Des. Q.*, núm. 66/67, p. 64-71, 1966, doi: 10.2307/4047334.

[4] «Hughes/Computer Graphics, 3/E [Book]». https://www.oreilly.com/library/view/hughescomputer-graphics-3e/9780133373721/ (consulta nov. 14, 2020).

[5] A. Appel, «Some techniques for shading machine renderings of solids», en *Proceedings of the April 30--May 2, 1968, spring joint computer conference*, New York, NY, USA, abr. 1968, p. 37-45. doi: 10.1145/1468075.1468082.

[6] J. T. Kajiya, «The rendering equation», *ACM SIGGRAPH Comput. Graph.*, vol. 20, núm. 4, p. 143-150, ago. 1986, doi: 10.1145/15886.15902.

[7] M. Pharr i G. Humphreys, *Physically Based Rendering: From Theory To Implementation*, vol. 2. 2004.

[8] P. Su, Q. Eri, i Q. Wang, «Optical roughness BRDF model for reverse Monte Carlo simulation of real material thermal radiation transfer», *Appl. Opt.*, vol. 53, núm. 11, p. 2324-2330, abr. 2014, doi: 10.1364/AO.53.002324.

[9] J. Burgess, «RTX On - The NVIDIA Turing GPU», *IEEE Micro*, vol. PP, p. 1-1, feb. 2020, doi: 10.1109/MM.2020.2971677.

[10] C. Dong, C. C. Loy, K. He, i X. Tang, «Image Super-Resolution Using Deep Convolutional Networks», *ArXiv150100092 Cs*, jul. 2015, Consulta: gen. 11, 2021. [En línia]. Disponible a: http://arxiv.org/abs/1501.00092

[11] E. Liu, «DLSS 2.0 – Image reconstruction for real-time rendering with Deep learning», p. 81.

[12] S. Bako *et al.*, «Kernel-predicting convolutional networks for denoising Monte Carlo renderings», *ACM Trans. Graph.*, vol. 36, p. 1-14, jul. 2017, doi: 10.1145/3072959.3073708.

[13] N. Thrane i L. Simonsen, «A Comparison of Acceleration Structures for GPU Assisted Ray Tracing», 2005.

[14] J. D. MacDonald i K. S. Booth, «Heuristics for ray tracing using space subdivision», *Vis. Comput.*, vol. 6, núm. 3, p. 153-166, maig 1990, doi: 10.1007/BF01911006.

[15] I. Wald, «On fast Construction of SAH-based Bounding Volume Hierarchies», en *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, USA, set. 2007, p. 33-40. doi: 10.1109/RT.2007.4342588.

[16] T. Möller i B. Trumbore, «Fast, Minimum Storage Ray-Triangle Intersection», *J. Graph. Tools*, vol. 2, ago. 2005, doi: 10.1145/1198555.1198746.

[17] M. Flynn, «Flynn's Taxonomy», en *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, p. 689-697. doi: 10.1007/978-0-387-09766-4_2.

[18] «CUDA Toolkit Documentation». https://docs.nvidia.com/cuda/ (consulta gen. 27, 2021).

[19] «Owczarczyk J (1988) Ray tracing: a challenge for parallel processing. Proc Parallel Processing for Computer Vision and Display, Leeds».

[20] B. Boehm, «A spiral model of software development and enhancement», *ACM SIGSOFT Softw. Eng. Notes*, vol. 11, núm. 4, p. 14-24, ago. 1986, doi: 10.1145/12944.12948.

[21] «The Stanford 3D Scanning Repository». http://graphics.stanford.edu/data/3Dscanrep/ (consulta març 06, 2021).