Ray Tracing

Ray Tracing یک روش برای رندرینگ در کیفیت بسیار طبیعی تر نسبت به Rasterization به صورتی ساده تر است. این روش تا به حال در صنعت بازی به صورت تجاری در امر رندرینگ استفاده نشده ولی روشی بسیار مناسب در رندرینگ های آفلاین و غیر هم زمان است . ولی با این رشد قوی سخت افزار ها و آمدن CPU های چند هسته ای آینده ای روشن در مقابل Ray Tracing قرار گرفته که البته این روش روش جدیدی نیست الان حدود 30 سال از معرفی این روش و سازماندهی فرمول های ریاضی اون میگذره که خیلی از آن ها در Rasterization هم کاربرد دارد. مطمینا چون این موضوع به صورت جامع در زمینه رندرینگ هم زمان مورد توجه نبوده در این زمینه پیشرفت خیلی خوبی نداشته ولی در آینده با به صرفه شدن این روش راه حل های بهتر و پیشرفته تر برای اون هم مورد توجه قرار میگیره و همیشه به این شکل فعلی نخواهد ماند.

بعد از یک مقدمه ی نه چندان خوب از Ray Tracing بریم سر اصل مطلب که اصلا چی هست!؟

Ray Tracing یک روش رندرینگ که از قوانین محیط اطراف ما برای رندرینگ استفاده میکنه یعنی تابش و باز تابش و چیزی که شما میبینید اثر باز تابش. به این صورت که مثلا شما در یک مختصات در محیط یک کره دارید حالا یک پرتو به این کره میخوره و با توجه به زاویه برخورد بازتاب میشه و اگر به چشم شما برسه شما آن را میبینید . این دقیقا کاریه که Ray Tracing انجام میده دو نوع Ray Tracing وجود دارد.

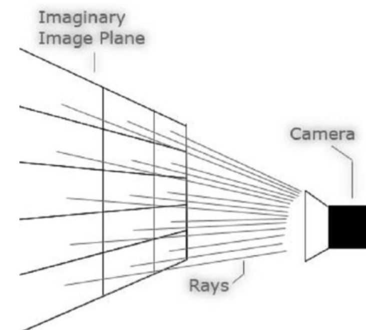
Forward RayTracing-1

در این روش پرتوها از یک منبع نور یا چندتا تابش میشن و به اشیا محیط برخورد میکنند اگر برخورد کنند! و بعد بازتاب میشند و اگر بازتاب اونها به دوربین فرضی ما بخورد اون نقطه رندر میشه و در صفحه نمایش نشون داده میشه این روش خیلی به صرفه نیست چون ممکنه خیلی از پرتو ها به دوربین ما نرسه (حداقل در مورد Offline Rendering رندرینگ غیر هم زمان) و بدیه دیگه ای که داره اینه که کیفیت تصویری که رندر میشه به شدت به میزان پرتو ها بستگی داره و تا بینهایت میشه تعداد را زیاد کرد تا به یک کیفیت نا مشخصی رسید!

روش بعدی رندرینگ RayTracing

Backward RayTracing-2

این روش خیلی خیلی برای کار ما به صرفه تر و روشی آینده دار تری هست! در این روش پرتوها از منبع نور تابش نمیشوند بلکه از دوربین به سمت محیط حرکت میکنند در این روش هر پرتو به سمت یک پیکسل تابش میشه در این روش دیگه از هر منبع نور چند ده هزارتا پرتو تابشه نمیشه و پرتو ها محدود به پیکسل ها بر روی صفحه فرضی ما میشه به شکل زیر:



خوب مشخصه که روش خیلی به صرفه تر و پرتو های اضافی به شدت کم شدند. اگر برای هر پیکسل یک پرتو در نظر بگیریم تعداد پرتو ها به تعداد پیکسل ها محدود میشه مثلا برای یک برنامه با رزولوشین 768 * 1024 ما نیاز به 786432 پرتو داریم در این روش پردازش به صورت زیر انجام میشه:

```
function Backward_RayTracing(scene)
```

```
{
foreach(pixel in image)
{
ray = Calculate_Ray(pixel);
closestObject = null;
minDist = "Some Large Value";
foreach(object in scene)
{
if(ray->Intersects(object) && object->distance <
minDist)
{
minDist = object->distance;
point_of_intersect = ray->Get_Intersection();
closestObject = object;
}
}
if(closestObject != null)
{
foreach(light in scene)
{
Shade_Pixel(point_of_intersect, light,
closestObject);
}
}
}
Save_Render("output.jpg");
}
```

و اینطوریا اگر معلوم نیست دقیقا فارسیش اینطوری میشه:

برای هر پیکسل حساب کن:

یک پرتو و کمترین فاصله که الان معلوم نیست یک عدد بزرگ باشه. کمترین فاصله منظور از یک شی در محیط و برای اینه که بدونیم کی پرتو را برای کدوم برخور را باید رندر کرد مثلا یک پرتو ممکنه در محاسبات ما اول با یک شی دورتر برخورد کنه و بعد در حلقه که داریم تست میکنیم به یک شی جلو تر بخوره اگر ما بخوایم اولین برخورد را رندر کنیم اول او دورتره رندر میشه! از این متغیر برای محاسبه نزدیک ترین استفاده میکنیم.

بعد برای هر شی:

چک میکنیم آیا پرتو به اون برخور میکنه یا نه. اگر برخورد کرد اون نقطه و اون شی را نگه میدارم و کمترین فاصله را هم همین قرار میدیم

بعد برای هر نور:

نسبت به نور های مختلف شی و نقطه برخور را سایه میزنیم. و بعد در نهایت نتیجه را در یک فایل ذخیره کردیم.

من یک سری چیز ها را نسبت به کد جا انداختم ولی خوب تو کد هست دیگه!

توجه داشته باشید که هر پیکسل با تمام شی ها در محیط برخوردش تست میشه. و البته مفهوم پرتو یک خط نیست بلکه یک بردار یکه است که یک نقطه شروع داره و یک جهت!.

اصل کار در RayTracing تشخیص برخورد هاست که همچین برای CPU ارزون تموم نمشیه چون هر پرتو باید با کلی چیز چک بشه و کد تشخیص برخورد هم هیچ وقت کد سبکی نیست شاید در آینده سبک به نظر بیاد!

خوب اجزا در محیط ما میتونند صفحه باشند یا کره یا مکعب یا مثلث یا هر شکل اولیه منظمه دیگه ای ... مهم نوشتن کد تشخیص برخورد پرتو با اونهاست که اگر بخواییم با گرافیک های امروزی و طرز ساختشون مقایسه کنیم همه چیز از مثلث ساخته شده و ما باید در تشخیص برخورد با مثلث خوب عمل کنیم که اتفاقا در RayTracing پردازش گیر ترینشون هم همینه!

من تو اینترنت گشتم تا یک کد ساده و کامل پیدا کنم راستش کمی ساده و کامل پیدا کردم ولی نه ساده و ساده! به هر حال کد های داخل یک کتاب را فکر میکنم از قبل داشتم البته فکر میکنم ! به هر حال تو پروژه ها پیداش کردم خیلی ساده و مناسب از رو همون توضیح میدم.

یک سری توضیحات کلی میدم که نمیخوام وارد جزییاتشون بشم

-کد برای نتیجه را روی مونیتور نشون نمیده بلکه تو یه فایل TGA ذخیره میکنه و هیچ چیزی نداره نه نور پردازی و سایه نه نه و نه هیچی فقط یک RayTracer خیلی ساده و خوب !

ما در اینجا یک بافر داریم که اونو پر میکنیم با رنگها بعدشم ذخیره میکنیم.

اول یک کلاس میسازیم برای تمام شکل های اولیه که برنامه پشتیبانی میکنه اسمشو میزاریم Shape که اینطوری میشه:

```
#pragma once
#include "Ray.h"
#include "Color3.h"
class Shape
{
public:

	Shape(void);
	virtual ~Shape(void);

	virtual	bool Intersect(const Ray &ray, float *dist)=0;

	virtual Color3 GetColor()
	{
		return color;
	}
	Color3 color;
};
```

همینطور که میبینید یا نمیبینید یک کلاس ساختیم که چیزایی که بقیه کلاس ها نیاز خواهند داشت را در بر داره مثل تابع intersect که تمام کلاس های اشکال اولیه ما دارند هر شکل دارای یک رنگ هست که میبینید . ما اینجا فاقد جنس و این جور مسایل هستیم و فقط یک رنگ میتونه هر شکل داشته باشه . برنامه ما از دو تا شکل اولیه دیگه پشتیبانی میکنه اولی صفحه و دومی کره است (Plane , Sphere) هر کدوم از این کلاس بالا به ارث میبرند . مهم ترین قسمتشون تابع Intersect که پایین برای کره را نشون دادم:

```
bool Sphere::Intersect(const Ray &ray, float *dist)
{
	// Get the ray to sphere vec and it's length.
	Vector3f rsVec = center - ray.origin;
	float rsLength = rsVec.dot(rsVec);

	// Does the ray pass through the sphere?
	float intersectDistA = rsVec.dot(ray.direction);

	// If not then no intersection with the ray.
	if(intersectDistA < 0 )
		return false;
```

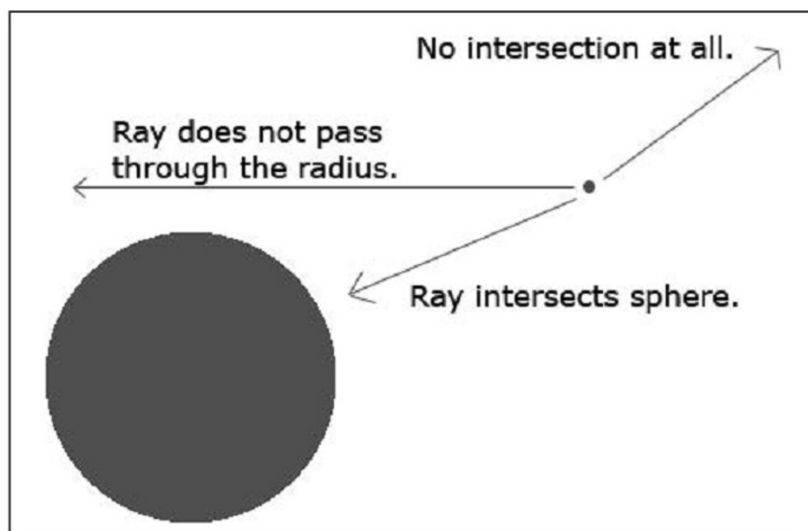
```
	// Does the ray fall within the sphere's radius?
	float intersectDistB = (radius * radius) - rsLength +(intersectDistA *
intersectDistA);

	// If not then no intersection with the ray.
	if(intersectDistB < 0)
		return false;

	// There is a hit so we record the results.
	if(dist != NULL)
	*dist = intersectDistA - (float)sqrt(intersectDistB);


return true;
}
```

خوب اینه! یه خورده ریاضی داره باید خودتون برید سراغش تو نت میتونید پیدا کنید سرچ کنید Ray Intersect یا Sphere Intersect و ... اینجور چیزا ولی در کل در ابتدا تست میکنه که اصلا پرتو به کره برخورد میکنه یا نه! اگر نه که اصلا دیگه کار را ادامه نده(با ضرب نقطه ای اینکار را میکنه)



ولی هنوز کافی نیست چون ما تا اینجا میتونیم بفهمیم که بردار وصل کننده پرتو به مرکز دایره با جهت پرتو زاویه 90 یا بیشتر داره یا نه ولی ممکنه زاویه کمتر از 90 داشته باشه ولی هنوز برخوردی نباشه برای اینکار از یک قایده مثلثی استفاده شده...

در نهایت اگر برخوردی باشه ما اونو در اشاره گر ورودی تابع نگه میداریم و true بازگشت میدیم یعنی با این شی برخورد صورت گرفته.

خوب در اینجا و این برنامه هر شکل یک رنگ داره پس مشکلی نیست ما فقط بگیم در چه فاصله ای به چه شکلی خورده بسه دیگه رنگش ثابت و مشخص!

RayTracer ما تابعی داره به نام Trace که کل شی ها را لیستشونو میگیره و برخوردشون را با پرتو ها تست میکنه اگر برخوردی باشه رنگ مورد نظر را بازگشت میده.

```
Color3 RayTracer::Trace(Ray &ray, vector<Shape*> &objList)
{
	// Closest intersected object and distance.
	float closestDist = 1000000;
	int closestObj = -1;

	// Loop through and find the closest intersection.
	for(int i = 0; i < (int)objList.size(); i++)
	{
		if(objList[i] == NULL)
		continue;

		// Intersection test.
		float dist = 0;
```

```
			// If the ray hits this object...
			if(objList[i]->Intersect(ray, &dist) == true)
			{
				// Record if this is the closest object.
				if(dist < closestDist)
				{
					closestDist = dist;
					closestObj = i;
				}
			}
		}

	// Return intersected object (if any).
	if(closestObj != -1)
	return objList[closestObj]->GetColor();

	// Return default color;
return defaultColor;
}
```

در هر دور نزدیک ترین شی برای پرتو مورد نظر محاسبه میشه (به ساده ترین شکل ممکن!).

و رنگ بازگشت داده میشه و در نتیجه در تابع Render گرفته میشه و در بافر ریخته میشه که بعدا در فایل TGA ذخیره بشه

```
bool RayTracer::Render(vector<Shape*> &objList)
{
	// Error checking...
	if(primaryBuffer == NULL)
		return false;

	// The primary ray and the current image pixel.
	Ray primaryRay;
	int index = 0;

	// Starting point for the ray.
	primaryRay.origin = Vector3f(0, 0, -600);

	// Generate image based on ray / object intersections.
	for(int y = 0; y < height; y++)
	{
		for(int x = 0; x < width; x++)
		{

			// Get the direction from the view to the pixel.
	primaryRay.direction = Vector3f(x - (float)(width >> 1),y -
(float)(height >> 1),depth);

			primaryRay.direction.normalize();

			primaryBuffer[index++] = Trace(primaryRay, objList);
		}
	}
return true;
}
```

و مثل اینکه اصل کار تموم شد در اینجا برای هر پیکسل محاسبات انجام میشه و یک پرتو تابش میشه و نتیجه در تابع Trace بازگشت داده میشه و در بافر ذخیره میشه کلا این سبک کار و روشی که اینجا بود اصل مطلب بود ولی خیلی خیلی ساده و ابتدایی نوشته شده و فقط برای فهم موضوع اصلا بازده خوبی نداره اصلا قابلیت های خوبی نداره و یک چیز کاملا ساده و قابل فهم فکر میکنم بادیدن کد و این نوشته ها بیشترشو بفهمید باتشکر از وقتتون .

امیر.

لینک مفید:

http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

یک نمونه عالی

http://homepages.paradise.net.nz/nickamy/simpleraytracer/simpleraytracer.htm

و همینطور در سایت DevMaster.net

http://www.devmaster.net/articles/raytracing_series/part1.php