Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

BAKALÁŘSKÁ PRÁCE

Plzeň, 2008 **Aubrecht Vladimír**

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Grafické efekty v počítačových hrách

Plzeň, 2008 Aubrecht Vladimír

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 30.4.2008 Aubrecht Vladimír

Abstract

Graphic effects in computer games

Reason for create this bachelors work is need to create fire into our game engine. My work contain search of existing methods for creating fire, explanation and implementation of particle based fire and light, which fire throw to the environment. Next in my work is engaged in implementation of this study to our engine and showing few used optimizations. Finish of my work is compare my results with real fire and light, which fire throw to environment. All my work expect with more fires and lights in game engine.

Obsah

1. Úvod

Důvodem pro vytvoření této bakalářské práce byla potřeba vytvoření efektu ohně do počítačové hry, kterou vyvíjíme spolu s dalšími kolegy.

Obsahem práce je prozkoumání existujících technik pro vytvoření efektu hořícího ohně, dále je zde teorie a implementace potřebná k vytvoření efektu hořícího ohně založeného na částicovém systému, implementace a popis osvětlení, které vrhá oheň a nakonec začlenění toho všeho do herního enginu s popisem a implementací optimalizací nutných pro plynulý běh. Předpokládá se, že ve scéně může být N nezávislých světel (na průměrné sestavě v kapitole měření výkonu se N pohybuje okolo 5ti viditelných ohňů). Nakonec práce ještě obsahuje kapitolu o tom, jak se dají vytvořit stíny, které oheň vrhá a porovnání dosažených výsledků s reálným ohněm a testy náročnosti na hardware.

Cílem práce je vytvoření funkčního herního enginu, kde by byly naimplementovány poznatky získané vypracováním této práce. Konkrétně se jedná o vytvoření herního enginu, který by uměl zobrazit N ohňů (konkrétně se jedná o pochodně) a plynule je renderoval na v dnešní době průměrné herní sestavě (upřesnění sestavy je k nalezení v kapitole s testy výkonosti).

2. Studium ohně

Vzhled ohně jsem studoval především z přiložených video nahrávek a fotografií. Jednotlivé metody by se daly rozdělit na realtime a non-realtime metody. Dále se již budu věnovat pouze realtime metodám, které jsou narozdíl od nonrealtime variant využitelné v počítačových hrách.

2.1.Metoda střídání textur

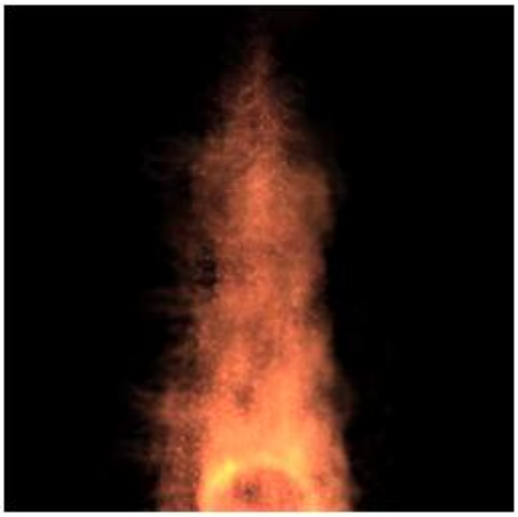
Pravděpodobně nejjednodušší metoda na implementaci. Princip metody spočívá v tom, že vykreslujeme billboard s texturou ohně. Tato textura s časem mění za jiné. Vlastní animaci ohně máme celou udělanou v texturách, které se pouze prohazují. Výsledný efekt je možný trošku vylepšit použitím více animací (ať už z nově udělaných textur, nebo těch samých) a tyto animace mezi sebou alpha-blendovat s použitím aditivního blendování. Je vhodné jednotlivé animace spustit různou rychlostí a po skončení animace znova spouštět z náhodného snímku.

Klady

- Jednoduchá implementace
- Rychlost renderingu (není nutné počítat žádné výpočty, jenom se kreslí textura)

Zápory

- Velká naročnost na paměť (je nutné nahrát dost textur)
- V případě použití standardní metody se animace stále stejně opakuje
- Oheň je vidět pořád ze stejné strany
- Neřeší kolize s předmětama



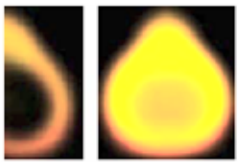
Oheň renderovaný metodou střídání textur

2.2. Perlin Fire (NVidia Corporation, 2008)

Tato metoda je velmi dobře zdokumentována a k nalezení jsou i zdrojové kódy. Výsledek této metody působí věrohodně (na první pohled k nerozeznání od pravého ohně).

Vytvoření toho efektu ohně spočívá ve vytvoření profilové textury, která určuje tvar, barvu a intenzitu ohně. Tato textura nemá žádná omezení, může být různě velká, atp. a záleží pouze na grafikovi, jakou udělá.

Obtočením této textury ve 3D prostoru podle osy Y získáme tzv. jednotkový tvar ohně. Rozměry objemu tohoto jednotkového ohně je [-0,5; 0,5] pro horizontální osy x a z a [0; 1] pro osu Y. Při tomto obtáčení provádíme aditivní směšování, čímž nám vznikne plamen. Tento plamen se zatím ještě nijak neanimuje a vypadá ze všech stran stejně.



V levo profilová textura a v pravo jednotkový tvar ohně

Vlastní animaci ohně provedeme tak, že spočítáme Perlinův šum pro čtyři dimenze (oheň je volumetrický, proto počítáme šum pro každou osu a čas), který aplikujeme na základní tvar ohně vytvořený profilovou texturou.

Princip této metody je velmi jednoduchý a proto se metoda může zdát snadná na implementaci. Potíž u tohoto efektu je ve vygenerování čtyřrozměrného Perlinova šumu pro každý pixel, což je velmi náročné na výpočet. Metodu je možné modifikovat a generovat jenom třírozměrný Perlinův šum, popř. jinou šumovou funkci, což přidá nějaký výkon na úkor kvality ohně.

Klady

- Oheň je plně interaktivní (lze modifikovat změnou parametrů)
- Oheň je volumetrický

Zápory

- Obrovská hardwarová náročnost (11 fps na nejvyšší detaily na ATI Radeon HD3870) Neřeší kolize s předměty

Perlin Fire

2.3. Metoda částicového systému (Microsoft Corporation, 2007)

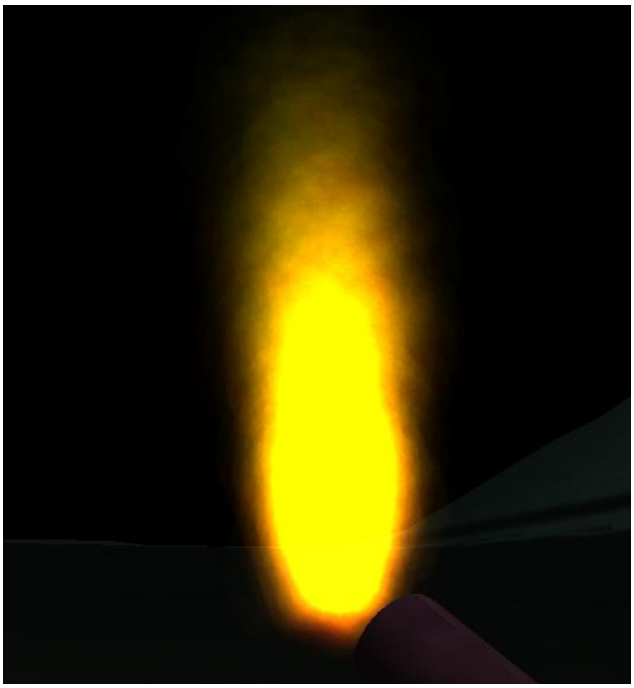
Metoda, kterou jsem si zvolil pro implementaci, proto tuto metodu v této kapitole pouze nastíním (podrobné vysvětlení metody nalezneme v dalších kapitolách). Jedná se o metodu, kdy vytvoříme částicový systém, kde částice mají nějakou velikost. Tyto částice vystřelujeme do vzduchu a po nějaké době je zabíjíme. Každá částice je natexturovaná jedinou připravenou texturou ohně a částice mezi sebou aditivně směšujeme. Každá částice má svojí dobu života, vektor rychlosti a svojí velikost.

Klady

- Dostatečně věrohodný efekt ohně
- Relativně malá náročnost na HW (lze naimplementovat s minimální náročností na cpu), bez problému lze renderovat 5 a více takovýchto ohňů v herní scéně na průměrné herní sestavě (podrobné specifikace lze najít v části s testy)
- Potřebné minimální množství zdrojů od grafiků (pouze 1 textura stačí)
- Ovlivňováním částicového systému lze měnit i celkem velký počet parametrů ohně (rychlost hoření, velikost a šířku plamene) a jednoduchou úpravou textury je možné měnit barevnost ohně, sytost, atp.

Zápory

- Implementace není nejjednodušší
- Jelikož je oheň založen na principu alpha blendování, tak je nutné používat při renderingu alpha blending, což má za následek snížení rychlosti vykreslování, protože nelze použít early z-buffer
- Oheň je vidět pořád ze stejné strany
- Neřeší kolize s předměty



Oheň generovaný částicovým systémem

2.4. Real-time Procedural Volumetric Fire(Institute of Data Analysis and Visualization (IDAV) and Department of Computer Science„ 2006)

Jedná se o metodu, která využívá křivek pro deformaci plamene a Perlinova popř. M-šumu pro generování vybrací u částic ohně. Oheň reaguje na předměty, které mu stojí v cestě a „obtéká je“. Ohni je také možno pomocí křivek jednuduše měnit tvar a docílit tak ohně vhodného pro libovolné použití (svíčka, táborák, lesní požár, ...)

Princip této metody je až po vygenerování tzv. jednotkového tvaru ohně shodný s metodou Perlin Fire. Dále se již metody liší. Narozdíl od metody Perlin Fire, která v této části začíná již aplikovat šum, této metodě jde o to, aby šel pomocí křivky modifikovat tvar ohně. Proto se vytvoří B-Spline křivka, která se využije k modifikaci tvaru ohně. Bohužel při modifikaci texturových koordinátů, pomocí křivky, zde vzniká problém tzv. inverzní parametrizace (problém, kdy je potřeba spočítat hodnotu, na základě které se počítá hodnota, kterou známe – nelze řešit analyticky pouze numericky).

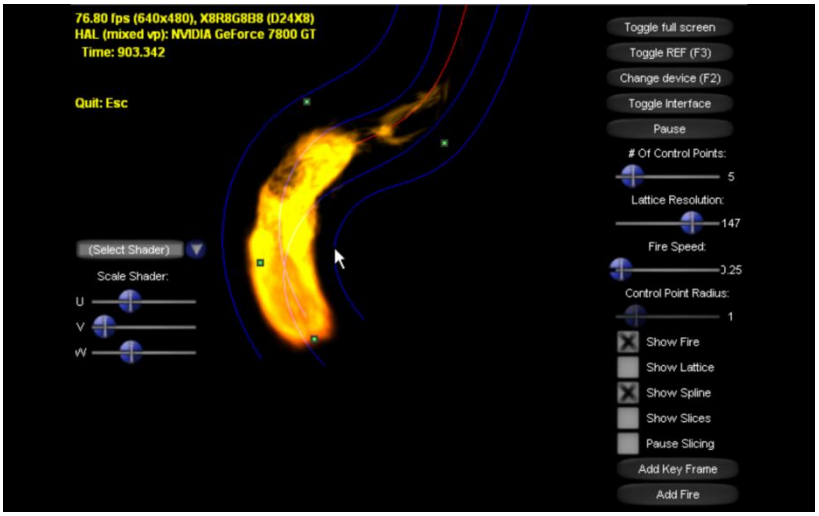
Díky již vyloženému principu jsme nyní schopni udělat zatím neanimovaný oheň, u kterého jsme schopni pomocí kontrolních bodů kontrolovat tvar (což nám umožňuje mimo jiné již lehce naimplementovat kolize s předměty). Dále se již pro vytvoření detailů ohně postupuje obdobně jako u metody Perlin Fire, tzn. šum se zanese do texturových koordinátů a tím provede posuny, které vytvoří detaily ohně. Rozdíl oproti metodě Perlin Fire je v tom, že se zde nepoužívá Perlinova šumová funkce, ale tzv. Improved Perlin Noise, popř. M-Noise funkce. Tyto šumové funkce jsou na rozdíl od Perlinovy funkce výrazně rychlejší a u funkce Improved Perlin Noise výsledek vypadá obdobně (Improved Perlin Noise je aproximací obyčejného Perlinova šumu).

Klady

- Velmi věrohodný efekt ohně
- Potřebné minimální množství zdrojů od grafiků (pouze 1 textura stačí)
- Je možné tvar ohně ovlivňovat jednoduchou změnou křivek
- Oheň reaguje na předměty, které mu stojí v cestě a obtéká je
- Oheň je volumetrický

Zápory

- Náročnost implementace



Realtime Procedural Volumetric Fire

3. Výběr vhodných metod a jejich implementace do herního enginu

Z důvodu minimálních hw nároků jsem si zvolil metodu založenou na částicovém systému. Metoda střídání textur je sice méně náročná pro cpu i gpu, ale má enormně velké nároky na paměť. Všechny výše zmíněné metody jsou určeny pouze pro simulaci vlastního plamene. K ohni samozřejmě patří i světlo, které vydává, proto se zde zaměřím na implementaci plamene, odpovídajícího světla k plamenu a začlenění obojího do herního enginu.

3.1. Implementace částicového systému pro efekt ohně

3.1.1. Datové uložiště

Každá částice částicového systému má svoje parametry, jako např. rychlost, velikost, atd. Částice si můžeme představit jako body – vertexy, u kterých jsme schopni nadefinovat potřebné parametry. Protože víme, že v jednom snímku se může vykreslovat více částic je vhodné umístit tyto vertexy do vertex bufferu, aby jsme byli schopni je rychle renderovat.

Momentálně víme, jak rychle vyrenderovat skupinu částic. Problém nastane, pokud budeme chtít nějaké částice odebrat (např. protože částice již dohořela), popř. přidat. Nejjednodušším způsobem je vždy přeuložit celý vertex buffer novými vertexy. Bohužel čím víc se odesílá vertexů do vertex bufferu, tím je operace časově náročnější, což si při renderingu nemůžeme dovolit (grafická karta by čekala na dokončení zápisu do vertex bufferu). Řešením je dynamický vertex buffer. Na rozdíl od statického vertex bufferu je u dynamického vertex bufferu možné provádět upload nových vertexů každý snímek.

Při standardním uploadu dat do vertex bufferu se čeká, než GPU dokreslí aktuální snímek a až poté se data odešlou na grafickou kartu, což pro naše potřeby není vhodné. Ideální by bylo, kdyby se data odesílali okamžitě. Toho lze docílit, že při uploadu vertexů do vertex bufferu nastavíme NoOverwrite flag, čímž GPU „slíbíme“, že data, která se kreslí, nebudeme přepisovat.

Zde bohužel narážíme na další problém. GPU a CPU jsou asynchronní a ve většině případů bude GPU zaneprázdněno kreslením ještě starého snímku, zatímco CPU už vypočítalo nový snímek. Mohlo by se stát, že pokud by nám nějaká částice již zemřela (dohořela) a my bychom na její místo ve vertex bufferu chtěli nahradit nějakou novou částicí, tak by se vyskytly synchronizační problémy, protože bychom přepisovali místo v paměti, u kterého jsme slíbili, že ho nebudeme přepisovat a GPU z něj stále může kreslit. Řešením tohoto problému může být počítání nových snímků. GPU by nemělo být nikdy více než dva snímky za CPU (toto nám zaručí ovladače grafické karty, kde se nachází toto omezení). Jednoduchou podmínkou lze tedy docílit toho, že na pozici ve vertex bufferu, zabranou nějakou částicí, která již zemřela, zapíšeme novou částici až po dvou snímcích a tím se těchto problémů zbavíme.

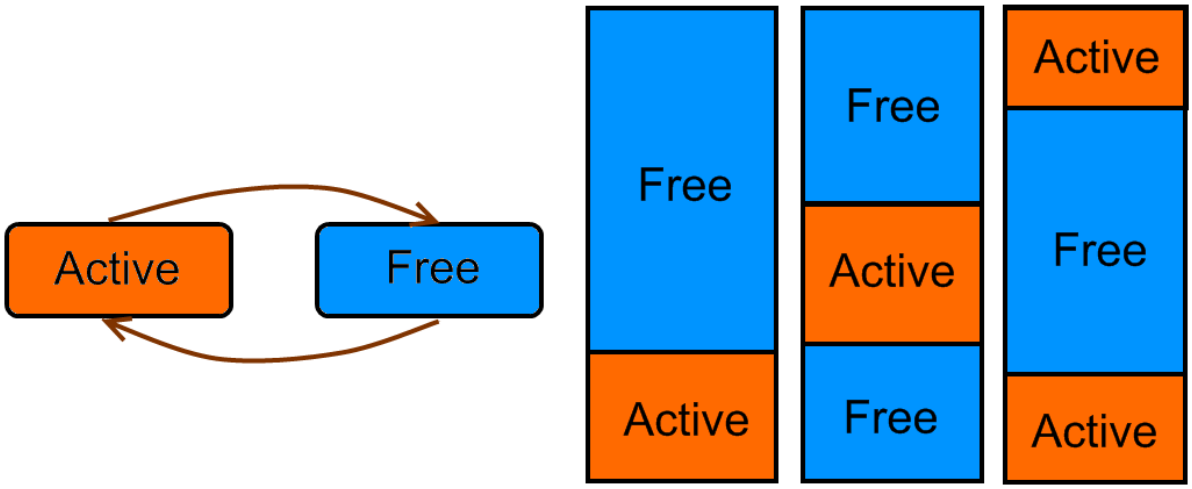
3.1.2. Práce s částicemi

Pod pojmem práce s částicemi si představíme přidávání a odebírání částic, protože v našem částicovém systému předpokládáme, že budou nějaké nové částice vznikat a i zanikat.

S těmito vlastnostmi narážíme hned na několik problémů. Některé z těchto problému již byly nastíněny při probírání datového uložiště (částici nelze odebrat okamžitě, protože jsme slíbili GPU, že vertexy, které jsou potřeba pro rendering, nebudeme přepisovat) a mezi ty ostatní patří třeba to, že máme omezenou velikost vertex bufferu, ale chceme kreslit neomezené množství částic.

Omezená velikost vertex bufferu

Jedna možnost, jak odstranit omezenou velikost vertex bufferu, je udělat **cyklický vertex buffer**.



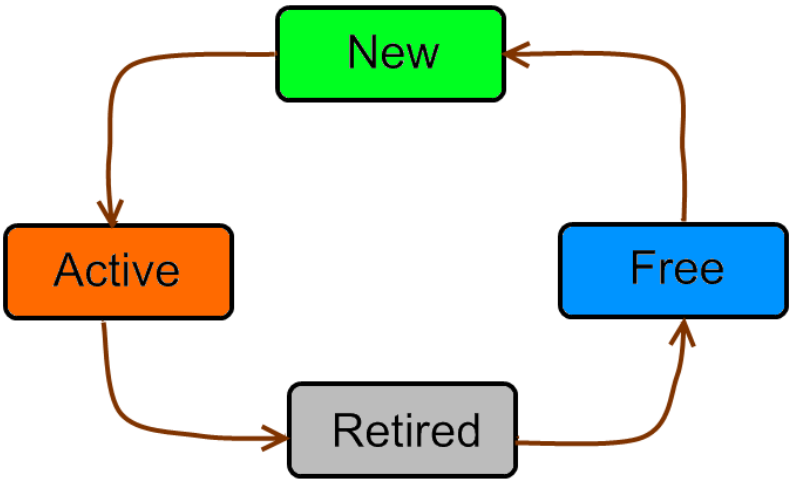
Cyklický vertex buffer

Na obrázku je možné vidět cyklický vertex buffer. Oblast **Active** v bufferu je oblast, kde jsou naše částice a naopak **Free** oblast, která je prázdná. Jak budeme částice Active přidávat, tak oblast Active se bude zvětšovat. Naopak jak částice budou umírat, tak oblast Active se bude zmenšovat. Nové částice přidáváme od začátku bufferu jednu po druhé, to nám zaručí, že víme, že nejstarší částice (tudíž částice, které mají umřít) jsou ve spodní části bloku Active. Na obrázku (cyklický vertex buffer) je možné vidět tři „stavy“ vertex bufferu. V prvním ze stavů jsme přidali pár částic. Druhý stav již ukazuje dobu, kdy nám po přidávání dalších částic už nějaké staré částice umřely. Nakonec třetí stav. Tento stav ukazuje cykličnost bufferu. Po tom, co nové částice dojdou ke konci bufferu a další není kam dávat, se začne znova od začátku, protože víme, že tam jsou již pouze mrtvé částice – tzn. je tam volné místo.

Řešení synchronizačních problémů(Microsoft Corporation, 2008)

Protože používáme dynamický vertex buffer a protože jsme mu nastavili flag NoOverwrite, tak si musíme ohlídat, které části vertex bufferu přepisujeme a s tím souvisí i problém zápisu do vertex bufferu.

V minulé části jsme si rozdělili vertex buffer na aktivní a volnou část. Aktivní část je část, kde máme umístěny vertexy, které se renderují. Bohužel s tímto rozdělením si vystačíme pouze pro rendering jednou naplněného vertex bufferu. Protože používáme dynamický buffer a flag NoOverwrite, nemůžeme si být jisti, jestli se data ještě kreslí, protože grafika může být s kreslením pozadu až o dva snímky. Z toho důvodu je nutné rozdělit aktivní část na část, kde jsou umírající částice (Retired – jedná se o blok, kde jsou již mrtvé částice, ale mohou se ještě kreslit, protože od jejich vykreslení ještě neuběhly již zmíněné dva snímky) a na část pro nové částice (New – jedná se o část, kam budeme odesílat nové částice, které se teprve začnou renderovat).



Kompletní přehled bloků vertex bufferu

Na obrázku je vidět, jak jdou jednotlivé bloky vertex bufferu za sebou. Toto pořadí bylo zvoleno, aby nám vždy zajišťovalo maximální souvislý blok vertexů a je tedy potřeba minimum volání metod pro vykreslení bufferu (ve většině případů pouze jedno volání, jediná vyjímka je, pokud je blok Active „roztržen“ na dvě části – na konci a začátku bufferu – v tomto případě jsou potřeba dvě volání).

Nyní si popíšeme nové bloky New a Retired. Blok New, jak již bylo řečeno, slouží pro přidání nových částic. Do tohoto bloku nahráváme vždy před započetím snímku nové částice (částice se odešlou na kartu okamžitě po tomto uploadu, protože je zde volání metody pro vykreslení primitiv, která data odešle). Počet částic může být maximálně takový, aby zabral celý blok Free.

U bloku retired vždy prohlížíme nejstarší částici, zda je alespoň dva snímky stará a pokud ano, tak zvětšíme blok Free na úkor bloku Retired (částice je určitě definitivně mrtvá).

Pseudokód ukazující vytvoření cyklického vertex bufferu, metodu Update zajišťující správnou aktualizaci jednotlivých bloků a metodu Render kreslící z tohoto bufferu

```
ParticleVertex[] particles;   //Pole vertexů
int firstActiveParticle; //Index v poli vertexů pro začátek Active bloku
int firstNewParticle;   //Index v poli vertexů pro začátek New bloku
int firstFreeParticle;  //Index v poli vertexů pro začátek Free bloku
int firstRetiredParticle; //Index v poli vertexů pro začátek Retired bloku
float currentTime;      //Aktuální čas v sekundách
int drawCounter;  //Počítá, kolikrát bylo zavoláno vykreslení


void AddParticle()      //metoda přidá částici do pole vertexů
{

      //posune se blok Free
      int nextFreeParticle = firstFreeParticle + 1;
      //pokud tento blok přetýká buffer, jde se od začátku
      if (nextFreeParticle >= particles.Length) nextFreeParticle = 0;
      //pokud další volná částice již sahá do bloku Retired, buffer je plný
      if (nextFreeParticle == firstRetiredParticle) return;
      //přidáme novou částici
      particles[firstFreeParticle] = newParticle;
      //aktualizujeme index první volné částice
      firstFreeParticle = nextFreeParticle;
}
void Update()     //metoda provádí aktualizaci jednotlivých bloků
{
      //provede aktualizaci aktuálního času
      currentTime += frametime;
      //zkontroluje stáří částic z bloku aktiv a pokud již částice mají být
      //mrtvé, tak jsou přesunuty do bloku Retired
      RetireActiveParticles();
      //hlídá, zda jsou částice z bloku retired nekresleny dost dlouho, aby
      //jsme si mohli být jisti, že už se nekreslí a když už se nekreslí,
      //jsou uvolněny
      FreeRetiredParticles();
      //v případě, že se nic nekreslí, tak se provede reset proměných
      //currentTime a drawCounter, podmínka je zdůvodu zabránění přetečení
      //currentTime
      if (firstActiveParticle == firstFreeParticle) currentTime = 0;
      if (firstRetiredParticle == firstActiveParticle) drawCounter = 0;
}

void Render()     //metoda se stará o renderování ohně
{
      //před začátkem snímku uploadnem nové vertexy do vertex bufferu
      If (firstNewParticle != firstFreeParticle) AddNewParticlesToVB();
      //když jsou nějaké aktivní částice
      if (firstActiveParticle != firstFreeParticle)
      {     //pokud jsou všechny aktivní částice ve spojitém bloku
            if (firstActiveParticle < firstFreeParticle)
            {     //proveď vyrenderování tohoto bloku
                  DrawPrimitives(firstActiveParticle, firstFreeParticle -
                              firstActiveParticle);
            }


            else  //částice nejsou ve spojitém bloku
            {     //vyrenderuj aktivní částice z konce bufferu
                  DrawPrimitives(firstActiveParticle, particles.Length -
                              firstActiveParticle);
                  //pokud jsou nějaké i na začátku bufferu
```

```
                    if (firstFreeParticle > 0)
                    {     //vyrenderuj aktivní částice ze začátku bloku
                          device.DrawPrimitives(0, firstFreeParticle);
                    }
              }

        }
        //zvýší se čítač kreslení
        drawCounter++;
}

//zkontroluje stáří částic z bloku aktiv a pokud již částice mají být
//mrtvé, tak jsou přesunuty do bloku Retired
void RetireActiveParticles()
{
        while (firstActiveParticle != firstNewParticle)
        {
        float particleAge=currentTime - particles[firstActiveParticle].Time;
        //pokud nejstarší částice ještě žije, žádně částice nepřesouvej
        if (particleAge < particleDuration) break;
        particles[firstActiveParticle].Time = drawCounter;
        firstActiveParticle++;
        if (firstActiveParticle >= particles.Length) firstActiveParticle = 0;
        }
}

//hlídá, zda jsou částice z bloku retired nekresleny dost dlouho, aby
//jsme si mohli být jisti, že už se nekreslí a když už se nekreslí,
//jsou uvolněny
void FreeRetiredParticles()
{
        while (firstRetiredParticle != firstActiveParticle)
        {
        int age = drawCounter - (int)particles[firstRetiredParticle].Time;
        //pokud nebyla nejstarší částice kreslena aspoň 2x, tak jí neuvolňuj
        if (age < 3) break;
        firstRetiredParticle++;
        if (firstRetiredParticle >= particles.Length) firstRetiredParticle=0;
        }
}

void AddNewParticlesToVB() //uploadne částice z bloku new do Vertex Bufferu
{     //Pokud je blok free spojitý
        if (firstNewParticle < firstFreeParticle)
        {     //uploadni vertexy z bloku Free do Vertex Bufferu
              SetVBData(particles, firstNewParticle, firstFreeParticle -
                        firstNewParticle);
        }
        else  //Pokud blok Free není spojitý
        {     //Uploadni vertexy z konce bufferu z bloku Free do VB
              SetVBData(particles, firstNewParticle, particles.Length -
                        firstNewParticle);
              //Pokud jsou nějaké vertexy v bloku Free i na začátku bufferu
              if (firstFreeParticle > 0)
              {     //Uploadni vertexy z bloku Free ze začátku bufferu
                    SetVBData(particles, 0, firstFreeParticle);
              }
         }
              //aktualizuj index první volné částice
              firstNewParticle = firstFreeParticle;
}
```

3.2.Implementace efektu ohně

V minulé kapitole jsme si vytvořili částicový systém, pomocí kterého teď budeme vytvářet efekt ohně. Při zkoumání ohně si můžeme všimnout, že při přidávání paliva do ohně bude oheň intenzivnější a naopak při docházení paliva bude oheň slábnout. V případě pochodně palivo samozřejmě nepřidáváme, ale po zapálení je pochodní zpalováno palivo z nádržky. Tuto vlastnost budeme očekávat i od našeho ohně, který má stejné chování při přidávání vertexů (protože se sčítají intenzity jednotlivých částic). Nádržku pochodně budeme simulovat konstantně rychlím přidáváním vertexů do našeho ohně (palivo u normální pochodně se spaluje přibližně stejnou rychlostí). Nakonec nesmíme zapomenout na odstraňování již dohořelých vertexů.

3.2.1. Popis částice

Nejprve si popíšeme, jak bude naše částice vypadat. Pro simulaci plamene musíme u každé částice znát její **pozici, rychlost** a **čas**, ve kterém začla částice hořet. Na základě těchto parametrů jsme schopni vypočítat novou pozici částice v jiných časech a zároveň určit, jestli by už částice neměla být mrtvá. Aby oheň získal na reálnosti a tolik se neopakoval je vhodné u každé částice vygenerovat **náhodná čísla**, podle kterých můžeme ovlivnit např. velikost, stáří, barevnost nebo rotaci částice.

Pozice

Jedná se o pozici, kde se bude nacházet oheň (tzn. pokud budeme chtít mít plamen např. na světových souřadnicích [0, 0, 0], tak všechny částice budou obsahovat tuto pozici). Jedná se o bod, ze kterého se emitují částice. Je samozřejmě možné tento bod měnit a nasimulovat tak třeba nějaký „hořící tvar“ (např. ohnivý kruh).

Rychlost

Určuje, kterým směrem a jak rychle se bude částice pohybovat. Výpočet je prováděn ve světových souřadnicích a proměnné **minHorizontalVelocity** a **maxHorizontalVelocity** jsou konstanty, které určují minimální a maximální možnou rychlost, kterou se částice může pohybovat vertikálně. Obdobný význam mají proměnné **minVerticalVelocity** a **maxVerticalVelocity**, které jsou pro vertikální směr letu.

Výpočet rychlosti se provede následujícím postupem (proměná **R** je náhodné číslo, které se generuje pro každý řádek výpočtu znova):

$$R \epsilon \langle 0; 1 \rangle$$

$$horizontalVelocity = (1 - R) \cdot minHorizontalVelocity + R \cdot maxHorizontalVelocity$$

$$velocity_x = horizontalVelocity \cdot \cos(2 \cdot \pi \cdot R)$$

$$velocity_z = horizontalVelocity \cdot \sin(2 \cdot \pi \cdot R)$$

$$velocity_y = (1 - R) \cdot minVerticalVelocity + R \cdot maxVerticalVelocity$$

Čas

Jedná se o čas, ve kterém byla částice vypuštěna z emitoru. Na základě tohoto času se později porovnává a počítá stáří částice.

Náhodná čísla

Bude se jednat o 3 náhodná čísla. Tato čísla budou určovat jak už bylo řečeno velikost, stáří a barevnost částice. Je vhodné tato tři náhodná čísla uložit jako 32 bitový integer (prvních 24b se obsadí třemi byte hodnotami, poslední byte bude prázdný), aby bylo možné je v shaderu přečíst jako jeden vektor.

3.2.2. Přidávání nových částic

Při zobrazení ohně většinou chceme, aby oheň hořel s přibližně stejnou intenzitou po celou dobu jeho hoření (rozhoření i zhasínání plamene se samozřejmě bude chovat mírně odlišně). Pro zajištění toho, aby oheň hořel, stačí před vlastním renderingem nahrávat do vertex bufferu nové částice, což nám bohužel nezajistí výše zmíněné chování, protože oheň bude závislý na počtu snímku za vteřinu, které počítač zvládá renderovat. Je to z toho důvodu, že vlastní rozjasňování a zhasínání ohně probíhá obyčejným aditivním blendováním více částic, tudíž, čím více částic, tím bude oheň jasnější a naopak. Této chyby se lze celkem jednoduše zbavit, pokud zavedeme odeslání částic do vertex bufferu v dávkách.

Pseudokód ukazující dávkování vertexů, aby oheň nebyl náchylný na velký počet snímků za vteřinu (bohužel neřeší problém s nízkým počtem snímků za vteřinu, což je ale pro naše potřeby jedno, protože oheň je určen do herního enginu a hra která má míň jak 25 snímků za vteřinu je nehratelná).

```
frametime = currentTime - lasttime; //výpočet doby jednoho snímku v ms

if (frametime > 20)     //po 20ti milisekundách
{
      count = frametime * 0.05;     //vypočtě se správný počet částic,
                                    //které mají být vytvořeny, očekáváme
                                    //jednu částici za jednu milisekundu,
                                    //protože víme, že tento blok má
                                    //proběhnout po 20ti milivteřinách, tak
                                    //násobíme (1/20)

      for (i = 0; i < count; i++)
      {
            particleSystem.AddParticles(baseNumberParticle);
      }

      lasttime = time;
}
```

3.2.3. Zpracování částic shaderem

Ve výše zmíněném textu jsme zatím pouze vytvořili systém, který umí renderovat na nějaké místo nějaké částice, které po čase zanikají. Tyto částice zatím vůbec nepřipomínají oheň ani vzhledem a ani pohybem. Klíčem k vytvoření tohoto efektu je využití vertex a pixel shader jednotek, pomocí nichž můžeme „říct“ každé částici, jak se má pohybovat a jak má vypadat.

Konstanty nutné k výpočtu ohně, které bude shader využívat

currentTime – Jedná se o aktuální čas

duration – Doba života částice

durationRandomness – Určuje náhodu v délce života částice (0=všechny žijí stejně dlouho)

gravity – Určuje použitou gravitaci (pro oheň je kladná v y složce, částice „padají vzhůru“)

endVelocity – Určuje, jak rychle se mění rychlost částice během jejího života (0 = na konci života se částice zastaví, 1 = na konci života částice dosáhne svojí přidělené rychlosti, čísla větší než jedná znamenají urychlení pro částici)

minColor – Určuje dolní mez barvy, kterou se bude přenásobovat barva texelu na částici (většinou bude sloužit pro nastavování počáteční hodnoty alpha kanálu, ale je možné modifikovat i ostatní barvy a měnit tak barevnost ohně vzhledem k jeho času života)

maxColor – Určuje horní mez barvy, kterou se bude přenásobovat barva texelu na částici(většinou bude sloužit pro nastavování konečné hodnoty alpha kanálu, ale je možné modifikovat i ostatní barvy a měnit tak berevnost ohně vzhledem k jeho času života)

startSize – Jedná se o velikost, jakou budou mít částice v době svého vytvoření (2D vektor, určující minimální a maximální hodnotu velikosti, velikost se bude určovat náhodně z tohoto rozmezí)

endSize – Jedná se o velikost, jakou budou mít částice na konci svého života (2D vektor, určující minimální a maximální hodnotu velikosti, velikost se bude určovat náhodně z tohoto rozmezí)

Věk částice

Pro všechny další výpočty nejdříve potřebujeme zjistit, jak je částice stará. Na základě této hodnoty jsme schopni dopočítat např. jak částice daleko vyletěla, popř. její barvu nebo ji můžeme využít k výpočtu rotace. Pro výpočet věku částice využijeme následujícího vztahu:

$$age = (currentTime - particleTime) \cdot (1 + R \cdot durationRandomness)$$

Pomocí tohoto vztahu zjistíme snadno, jak částice dlouho žila, víme aktuální čas **currentTime** a čas, ve kterém byla částice vypuštěna **particleTime**. Druhá závorka ve vztahu nám vnáší náhodnost do života částice. Proměná **R** zde vystupuje jako první z náhodných čísel, které jsme v minulé kapitole generovali ke každé částici a proměná **durationRandomness** je koeficient, který určuje, jak moc ovlivňuje náhoda život částice. Nakonec ještě provedeme normalizaci věku, díky čemu zjistíme, kdy má být částice již mrtvá (po věku 1 by měla částice zemřít). Pro normalizaci využijeme toho, že víme, jak dlouho by měla částice žít.

$$normalizedAge = saturate\left(\frac{age}{duration}\right)$$

Kde funkce saturate je definována

$$saturate(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 1 \\ x, & x \in \langle 0; 1 \rangle \end{cases}$$

Pohyb částice

Nejprve si spočteme dráhy částice. Víme dráhu, která bude přímočará ve směru vektoru zadané rychlosti. Můžeme použít skalární výpočet výsledné dráhy, protože počáteční rychlost částice a konečnou rychlost částice, je možné určit rychlost částice podle vzorce:

$$v = v_0 + (v_1 - v_0) \cdot t$$

Kde v_0 je počáteční rychlost částice, v_1 je konečná rychlost částice a t je čas, ve kterém chceme spočítat rychlost částice. Počáteční rychlost $\mathbf{v_0}$ vypočteme jako velikost našeho vektoru rychlosti částice (**velocity**). Konečnou rychlost vypočteme podle vzorce:

$$v_1 = v_0 \cdot endVelocity$$

Pro výpočet velikosti dráhy nám tedy chybí už pouze poslední proměnná a to je **čas**, za který dosadíme náš **normalizovaný věk.** Z fyziky víme, že po integraci rychlosti podle času dostáváme dráhu.

$$\int v\,dt = \int (v_0 + (v_1 - v_0)t)dt$$

$$s = v_0 t + (v_1 - v_0)\frac{t^2}{2}$$

Nyní, když víme velikost uražené dráhy, tak jsme schopni vypočítat novou pozici částice. Víme startovní pozici částice, víme, jak dlouho letěla, a podle vektoru rychlosti, který si u sebe částice nese, víme i její směr, proto nová pozice bude:

$$position = position + normalize(velocity) \cdot s \cdot duration$$

Kde funkce normalize je definována:

$$normalize(\vec{x}) = \frac{\vec{x}}{\sqrt{x_x^2 + x_y^2 + x_z^2}}$$

Nakonec nám ještě zbýva do výsledné pozice započítat vliv gravitace, kterou započítáme pomocí vzorce:

$$\overrightarrow{position} = \overrightarrow{position} + \overrightarrow{gravity} \cdot age \cdot normalizedAge$$

Velikost částice

Oproti výpočtu nové pozice částice je výpočet velikosti dosti jednoduchý. Celý výpočet spočívá pouze v tom, že protože chceme vnést trošku náhodnost do velikosti částice, tak nejprve spočteme lineární interpolaci v rozmezí počáteční a konečné velikosti přes druhé námi vygenerované náhodné číslo pro částici, označíme si ho zde $\mathbf{R_S}$.

$$startSize = (1 - R_S) \cdot startSize_x + R_S \cdot startSize_y$$

$$endSize = (1 - R_S) \cdot endSize_x + R_S \cdot endSize_y$$

Zde **vektory startSize** a **endSize** jsou naše konstanty, které určují rozpětí počáteční a konečné velikosti a **scaláry startSize** a **endSize** jsou již námi vypočítané konkrétní hodnoty pro velikost konkrétní částice.

Nyní nám již pouze stačí spočítat „proměnu“ velikosti částice na základě jejího stárnutí. To provedeme opět lineární interpolací na základě normalizovaného času.

$$size = (1 - normalizedAge) \cdot startSize + normalizedAge \cdot endSize$$

Barva částice v závislosti na délce života

V této kapitole se dozvíme, jak vypočítat koeficient pro přenásobení barvy texelu textury ohně částice tak, aby nově vytvořené částice se postupně objevovaly (sílila jejich intenzita, jakoby začínal hořet plamínek) a s koncem svého života pomalu mizely (slábla jejich intenzita jakoby vyhasínaly). Pro začátek začneme tím, že vneseme zase trošku náhodnosti mezi jednotlivé částice, tak si z rozsahu, který máme definovaný konstantami **minColor** a **maxColor** vypočítáme na základě třetího náhodného čísla **R** základní barvu, kterou budeme ovlivňovat texel.

$$\overrightarrow{color} = (1 - R) \cdot \overrightarrow{minColor} + R \cdot \overrightarrow{maxColor}$$

Nyní potřebujeme, aby se částice rozsvěcovala a pohasínala. Toho docílíme změnou alpha kanálu. Na následujícím grafu vidíme funkci:

$$f(x) = x \cdot (1 - x) \cdot (1 - x)$$

Přenásobenou různými koeficienty. Tato funkce f(x) se ukázala jako vhodná pro modifikaci alpha kanálu, protože jako u ohně je zde rychlé rozjasnění a pomalý útlum intenzity.

Proměnná x nám představuje **normalizovaný čas** a hodnota funkce představuje velikost složky alpha kanálu (intenzitu ohně).

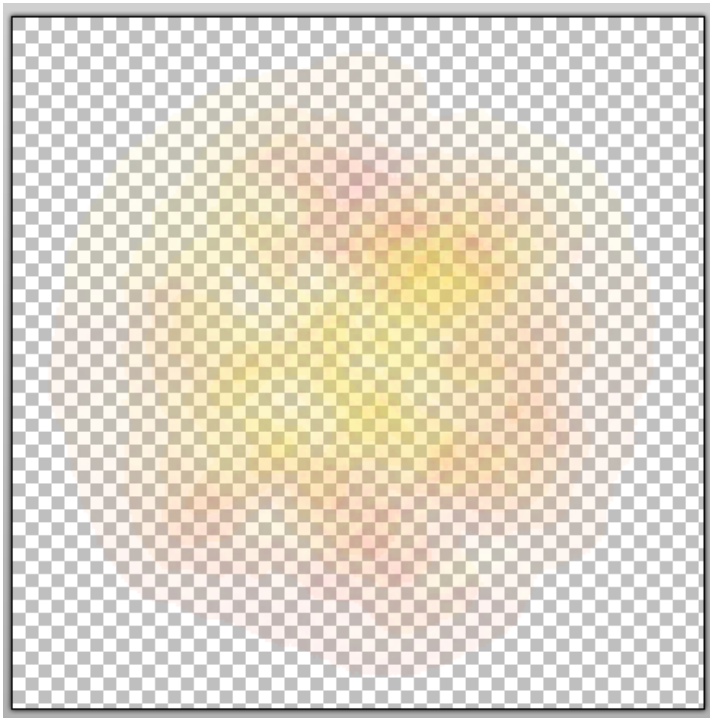
Původní funkce se bohužel pouze blíží reálnému rozjasňování a pohasínání, je proto vhodné tuto funkci ještě přenásobit konstantou, čímž můžeme zrychlit, popř. zpomalit rozjasňování a vyhasínání ohně. Při testování jsem zjistil, že nejblíže reálné hodnotě má funkce, když je přenásobená koeficientem 30.

Výsledná barva částice je nakonec určena barvou texelu z textury přenásobená touto vypočítanou barvou.

3.2.4. Textura ohně

Jedná se o texturu, kterou se budou texturovat jednotlivé částice. Výsledný efekt ohně se tvoří aditivním směšováním více částic, které mají různou velikost, rychlost a jas (je určen alpha kanálem). Textura určuje pouze zbarvení plamene a intenzitu jednotlivých částic. Alpha kanál určuje základní jas, který je přenásoben barvou vypočítanou z minulé kapitoly (ta určí výsledný jas a barvu částice, pokud se přenásobí alpha kanál textury nulou, částice zmizí).

Barva textury zase určuje zbarvení plamene. Oheň pochodně má barva oranžovo-žluto-bílou. Na okrajích je trošku do ruda a ve středu trošku do bíla. Bíla barva v textuře není, protože tím, že máme jeden emitor, ze kterého vylétají částice, tak nám jde hodně částic přes sebe a aditivním směšováním se z barvy, která je vidět na obrázku textury stane bílá.



Textura použitá pro texturování částic ohně (na čtverečkovaném pozadí kvůli viditelnosti průhlednosti)

3.3. Implementace světla a integrace do herního enginu

V herním enginu je potřeba, aby bylo možné mít ve scéně N světel. Pro tyto účely se ukázalo vhodné použít techniky „**Light per pass**“. Jedná se o techniku, kdy postupně renderujeme celou scénu vždy s aktivním jedním světlem (je možno modifikovat a zpracovávat světla třeba po třech) a výsledné rendery aditivně směšujeme. Technika získala toto jméno proto, že se implementace provádí tak, že pro každé světlo provedeme jeden průchod shaderem, který scénu vyrenderuje nasvícenou momentálně aktivním světlem.

3.3.1. Implementace bodového světla ohně

Světlo ohně jsem implementoval jako bodové světlo, protože oheň svítí do všech směrů, stejnou intenzitou a s rostoucí vzdáleností klesá jeho intenzita. Pro útlum intenzity s rostoucí vzdáleností jsem použil vzorec:

$$Attuentation = \frac{1}{1 + 0.0004 \cdot D + 0.000028 \cdot D^2}$$

Kde **D** je vzdálenost vertexu od ohně. Koeficienty byly získány pokusným nastavováním a ověřováním reálnosti dosahu světla. Protože oheň, jak je vidět na natočeném videu „bliká“ je zde ještě provedena simulace blikání. Toto blikání bylo provedeno tak, že při každém snímku se do shaderu nahrají čtyři náhodná čísla (R1, R2, R3, R4) a intenzita se přenásobí koeficientem blikání. Čtyři náhodná čísla byly použitá, protože průměr více náhodných čísel nám zaručí větší podobnost zprůměrovaných náhodných čísel a méně extrémních hodnot, proto oheň nebude působit tak blikavým dojmem. Koeficient se spočte následovně:

$$k_b = (R_1 + R_2 + R_3 + R_4) \cdot 0.25 \cdot (1 - m) + m, \qquad m \in \langle 0; 1 \rangle$$

Kde **m** určuje minimální intenzitu světla, kterou chceme během blikání (0 říká, že světlo bude pořád zhaslé a jednička naopak, že světlo bude pořád svítit). Vzorec byl odvozen ze vzorce pro lineární interpolaci. V našem vzorci máme konstantní parametr **m** a danou maximální svítivost ohně (v rovnici se jedná o přenásobení jedničkou v posledním členu rovnice).

3.3.2. Light per pass(GameDev.net, 1999)

Tuto techniku jsem zvolil z toho důvodu, že umožňuje ve scéně zobrazit N světel (kde N je limitováno pouze hardwarem, na kterém aplikace běží – na běžné herní sestavě viz. níže se toto N pohybuje okolo pěti viditelných světel) a zároveň lze tuto metodu naimplementovat s relativně nízkými HW nároky, což je pro herní engine důležité. Technika využívá toho, že RGB systém je aditivní a pro přidání světla do scény se scéna vyrenderuje nejprve pouze otexturovaná, tzn. nenasvícená. Po tomto prvním renderingu se do scény „přidává světlo“. Scéna je po prvním renderingu uložená ve frame bufferu, pomocí něhož můžeme provádět alpha blending.

Vzorec pro alpha blending:
FinalColor = SourceColor * SourceFactor + DestinationColor * DestinationFactor

Ze vzorce je patrné, že k výslednému vyrenderovanému obrazu je možné přičíst další barvu pomocí alpha blendingu. Při míchání světel dochází také ke sčítání barev v RGB systému, tak toho využijeme a provedeme druhý render, ve kterém vyrenderujeme scénu otexturovanou a

nasvícenou prvním světlem. Pro tento render je nastaven aditivní alpha blending s *SourceFactorem* a *DestinationFactorem* rovným jedné, protože oba rendery mají stejnou váhu. Čímž dostaneme, že výsledná barva bude:

FinalColor = material + material * LightIntenzite = material * (1 + LightIntenzite)

Kde *material* je barva pixelu vybraná z textury a *LightIntenzite* je intenzita našeho světla. Po vytknutí materiálu je vidět, že se nám ve vzorci objevila v závorce jednička. Tato jednička představuje ambientní složku světla. Ovlivnění této složky je možné tím, že provedeme první render rovnou s ambientní složkou.

Tímto jsme do scény přidali první světlo. Stejným postupem je možné přidat dalších N světel. Bohužel tato metoda je velmi náročná na fillrate, čímž je tedy převážně limitován počet světel.

Další nevýhoda metody je, že se scéna musí renderovat N-krát, což vlastně znamená N-krát uploadovat všechny data (vertexy, indexy, textury, matice, atd.) na kartu. Dále je zpracovat vertex shader jednotkou a všechna primitiva znova narastrovat. Tato nevýhoda se dá naštěstí částečně odstranit.

3.3.3. Distance optimize light per pass

Princip metody zůstává stejný. V čem se metoda liší je to, že pokud víme, že některá místa nejsou nasvícená světlem (pro který momentálně děláme rendering), tak tato místa budou v renderu pro světlo (pro který render provádíme) vyrenderována černě. Černá barva odpovídá RGB vektoru (0, 0, 0).

Po dosazení do vzorce:

$$\overrightarrow{finalColor} = \overrightarrow{material} + (0,0,0) = \overrightarrow{material}$$

Zjistili jsme, že pro takto renderované objekty se výsledná barva **nezmění.** Což znamená, že nemá smysl takovéto objekty vůbec renderovat.

Optimalizace má samozřejmě smysl jenom pro světla, která mají omezený dosah. Protože světla mají omezený dosah (který víme), tak jsme schopni jednoduše rozhodnout, zda se objekt nachází v dosahu světla, či ne. Toto lze provést např. tak, že při renderu objektu pro konkrétní světlo spočteme vzdálenost mezi objektem a světlem a pokud je tato vzdálenost větší než dosah světla, tak objekt není nasvícen a můžeme přeskočit jeho rendering.

Díky takto jednoduché optimalizaci se celý rendering (obzvlaště s více světly a s více objekty) několikanásobně zrychlí. Bohužel tato metoda je nápomocná **pouze pokud nejsme limitování fillratem.** Výhoda této metody se projeví až v kombinaci s další zmíněnou optimalizací.

3.3.4. Z-Buffer optimize light per pass(AMD Corporation, 2008)

Díky této metodě jsme schopni **podstatně snížit fillrate** a vůbec celkovou zátěž grafické karty výpočty a zatížením sběrnice. Metoda spočívá v tom, že v jednom snímku je scéna statická, tzn. scéna je statická pro rendering každého světla. Díky tomuto i víme, že Z hodnoty jednotlivých primitiv budou identické s předchozím renderem a jednoduchým nastavením porovnávací funkce Z-Bufferu na **Equal** před renderingem jednotlivých světel můžeme lehce zajistit, že většina pixelů bude zamítnuta díky early z-buffer optimalizaci.

V kombinaci s minulou optimalizací se technika Light per Pass stává dostatečně efektivní na použití v herním enginu. Samozřejmostí pro použití v herním enginu je, aby u jednotlivých objektů byla zjišťována viditelnost a na základě té proběhl/neproběhl rendering. V odevzdané práci je pro zjišťování viditelnosti použita bounding sphere objektu, se kterou se provádí detekce kolize s pohledovým jehlanem (view frustum). Tato metoda se ukázala jako nejrychlejší, protože pro test je nutné provádět pouze jeden skalární součin, narozdíl od detekce kolize s bounding boxem, kde jich je potřeba osm (přesnost bounding boxů – tzn. přesněji vyhodnotíme, zda je objekt viděn, či ne - v tomto případě nenahradí ztrátu výkonu, kterou dostaneme použitím bounding boxů).

Algoritmus na detekci kolizí s bounding sphere (pseudocode)(Gamedev.net, 2008)

```
bool ComputeVisibility()
{

        forech (Plane p in FrustumPlanes)
        {

                float dot = p.Dot(objectPosition + boundingSphereCenter);
                if ( dot + boundingSphereRadius < 0)
                        return false;

        }

        return true;

}
```

3.3.5. Další optimalizace

Samozřejmě stále platí, že čím méně posíláme do grafické karty vertexů, tím rychleji se scéna renderuje (provádí se méně výpočtů a nezatěžujeme tolik sběrnici). Proto je vhodné používat další optimalizace jako **Level of Detail**, které nám podstatně srazí počet trojúhelníků a které se nechají celkem snadno naimplementovat (ve vyvýjeném enginu se ukázala jako dostatečná třída ProgressiveMesh z Direct3D, která je má podporu Level of Detail).

Další velmi efektivní optimalizací je seřazení objektů tak, aby po sobě šly vždy objekty, které používají stejný shader. Tím minimalizujeme počet uploadů shaderu do grafické karty, které velmi ovlivňují výkon. Bohužel tato optimalizace jde proti optimalizaci z kapitoly *Distance optimize per pass*, ale je možné je zkombinovat (např. třídit podle shaderu a pak až podle vzdálenosti).

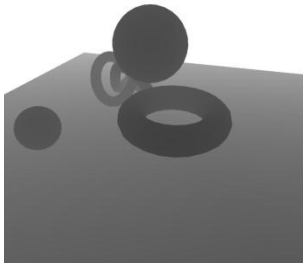
Nakonec optimalizace, kterou je prakticky nutné udělat, protože bez ní je srážen výkon extrémně. Je nutné používat multitextury a texture mapping. Pro programátora to není žádná práce navíc, ba naopak. Bohužel grafikovi se tím přitíží značně. Vliv na výkon to má ale obrovský, protože to znamená, že pro každý objekt je nutné nahrávat pouze jednu texturu (místo třeba patnácti). Účinnost této optimalizace navíc roste N krát právě při renderingu světel, protože pro rendering každého světla se do shaderu nahrává pouze jedna textura místo více (což znamená, že např. pro objekt s 10 texturama by se ve scéně se třemi světli nahrávalo do shaderu 30 textur zatímco při použití multitextury se budou nahrávat pouze tři).

3.4.Stíny vržené ohněm (NVidia Corporation, 2004)

Hořící oheň kromě toho, že vrhá světlo, tak vrhá vytváří i stíny. Jednoduchou metodou pro vytvoření stínů, které vrhá světlo je vytvoření stínů pomocí techniky shadow mapping. Bohužel tato metoda sama o sobě nestačí na vytvoření stínů od všesměrového světla, je vhodná pouze pro směrová světla. Tato omezení se dají vyřešit použitím metody Omnidirectional Shadow mapping, která rozšiřuje původní shadow mapping právě o možnost vytváření stínů pro všesměrová světla.

3.4.1. Shadow mapping

Začneme vysvětlením principu jednoduchého shadow mappingu, protože pochopení této techniky je základ v pochopení Omnidirectional Shadow mapping, která na této metodě staví. Technika shadow mapping spočívá ve vytvoření tzv. stínové mapy (shadow map), ve které jsou vyrenderovány všechny objekty a v barvě, kterou jsou vyrenderovány, je zanesena vzdálenost od světla.



Ilustrační ukázka stínové mapy

Na základě této stínové mapy se poté provede rendering klasické scény, kde se při renderingu rozhoduje zda vzdálenost pixelu objektu od světla, kterou zjistíme ze stínové mapy, je větší nebo rovna vzdálenosti pixelu objektu od pixelu v prostoru světla, který právě zpracováváme. Pokud podmínka platí, tak pixel není ve stínu, v opačném případě je ve stínu.

Touto velmi jednoduchou metodou jsme schopni říct, kde je stín a kde není. Bohužel pro výpočet matice, která nám definuje prostor světla jsme nucení znát pozici světla a směr, kterým se světlo dívá, což u všesměrového světla není možné určit. V tomto nám pomůže následující metoda.

3.4.2. Omnidirectional Shadow mapping

Tato metoda staví na minulé metodě a rozšiřuje ji o možnost využití při použití všesměrového světla. V předchozí metodě jsme si řekli, jak vygenerovat jednu shadow mapu v nějakém směru. U všesměrového světla chceme stíny ve všech směrech, což je vlastně 6 směrů (+x, -x, +y, -y, +z, -z). Tato metoda využívá toho, že známe tyto směry a provede render šesti stínových map do krychlové textury (CubeTexture). Tato textura si lze představit jako krychle, jejíž stěny vždy obsahují jednu texturu (v našem případě stínovou mapu).

Vlastní rendering se poté provede obdobně jako u techniky Shadow mapping. Jediný rozdíl oproti této metodě je, že určení hodnoty ze stínové mapy se provede na základě vektoru směru od světla ke zpracovávanému vertexu, který se nachází v prostoru aktuálně zpracovávaného světla.

4. Ověření věrohodnosti efektu ohně a měření výkonu

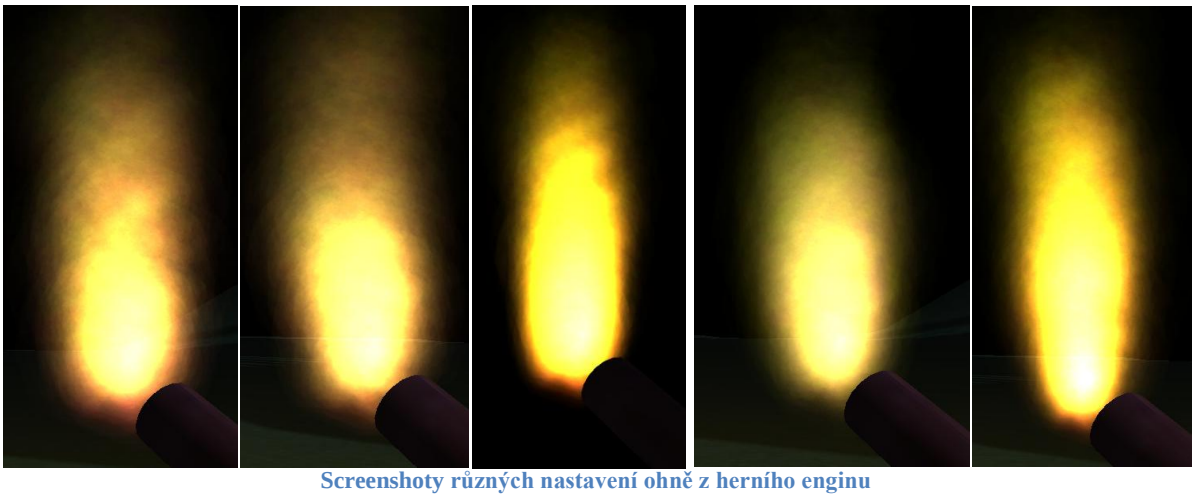
Při ověřování věrohodnosti ohni jsem vycházel především z video nahrávek, které jsem pořídil. Soustředil jsem se především na vzhled plamene a na světlo, které vrhá. Oheň byl natáčen v zatemněném sklepě, v malé místnosti, aby bylo dobře vidět světlo na stěnách.

4.1.Vzhled plamene

Celý vzhled generovaného plamene lze nastavit pomocí konstant, které jsem uvedl v kapitole zpracování částic shaderem, popř. Rychlost rozjasňování a zhasínání plamene lze upravit modifikací křivky, která je zmíněná v téže kapitole.

Jako referenční oheň jsem použil video ohně mnou natočeného (je obsaženo v příloze). Přikládám pár náhledů na originální plamen a na generovaný plamen.

Oheň natočený videokamerou (první tři) a dvě fotografie nalezené na internetu

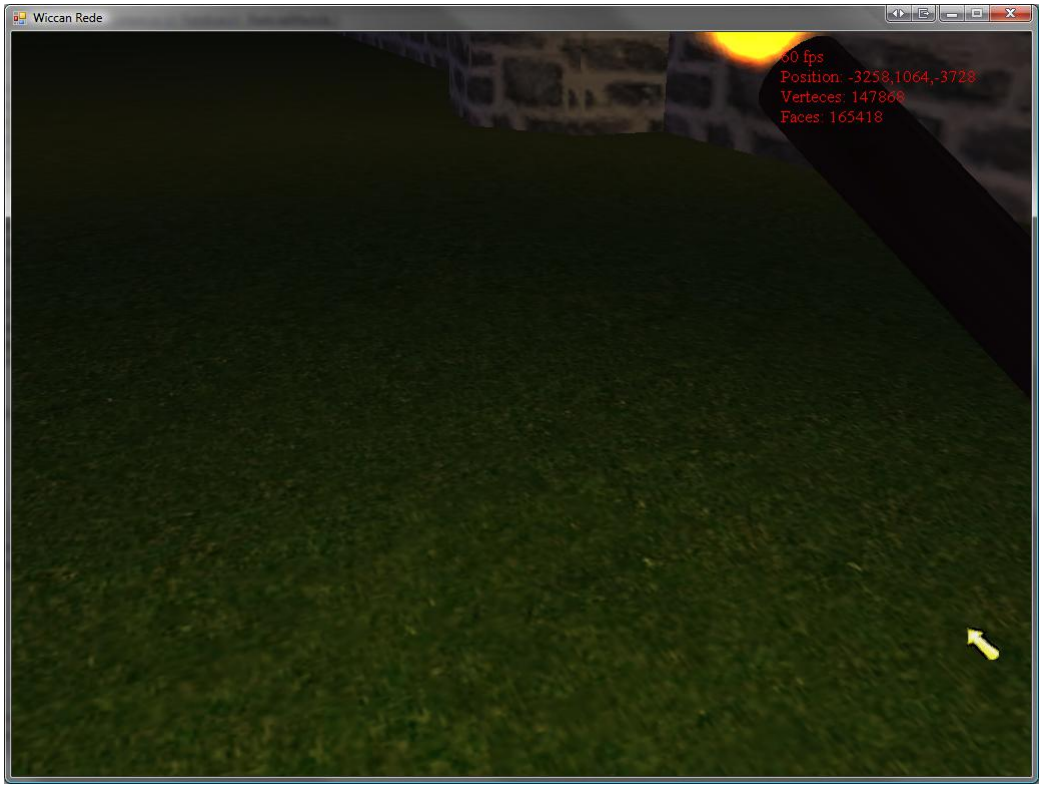


Screenshoty různých nastavení ohně z herního enginu

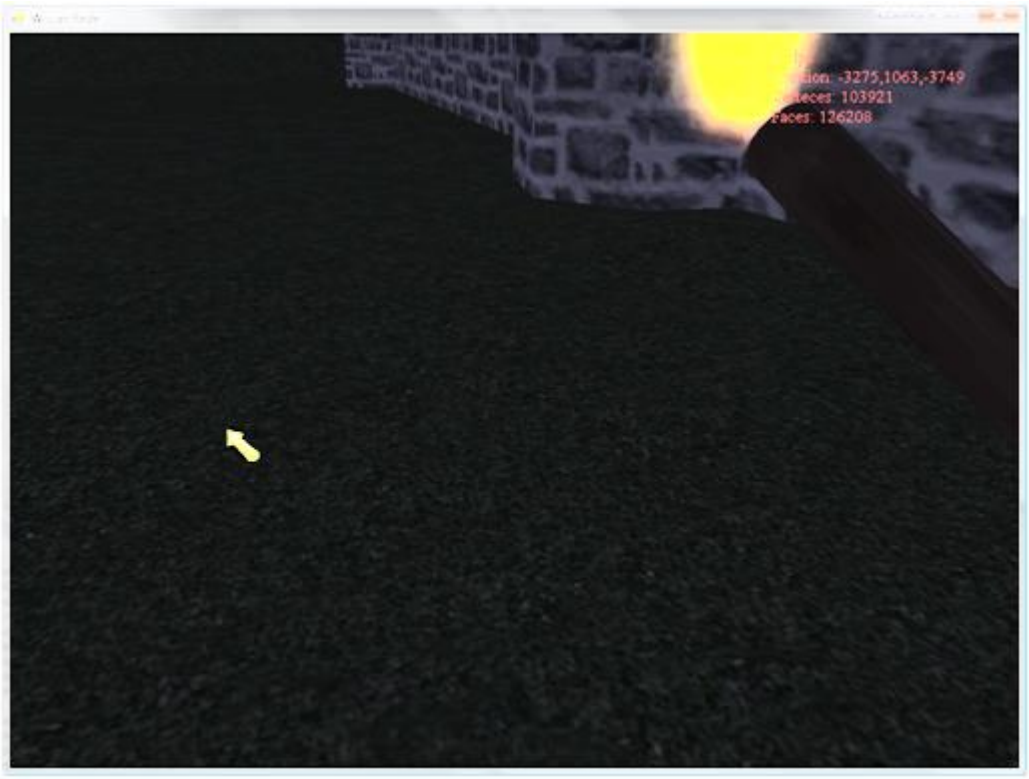
Bohužel na screenshotech toho není moc vidět. Mnohem více detailů a hlavně vlastní animace je vidět na přiloženém videu a ve herním enginu.

4.2.Světlo, které oheň vyzařuje

Na přiloženém videu je vidět, že místnost osvětlovaná ohněm jakoby bliká, proto jsem se pokusil o simulaci blikání bodového světla, kterým oheň je. Na obrázcích níže je vidět nasvětlená scéna pochodní, kterou hráč drží v ruce a stejná scéna, kde je vypnuto světlo od pochodně (v obou scénách se nachází v této části ještě mírně namodralé světlo od měsíce).



Scéná s aplikovaným světlem od pochodně



Scéna bez světla od pochodně

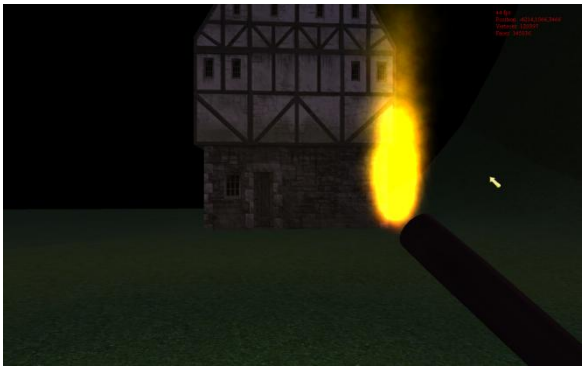
Bylo použito nažloutlé světlo, proto je terén na prvním obrázku zbarven do žluta. Pro blikání je bohužel nutné spustit aplikaci, protože ho není možné ukázat na statických obrázcích (blikání je jemné, na papíře by bylo téměř nerozpoznatelné, když bych umístil dva obrázky vedle sebe).

4.3.Měření výkonu

Měření jsem prováděl na dnešní běžné herní počítačové sestavě:

Základní deska	Asus P4C800-E Deluxe
Procesor	Intel Core 2 Duo E6300 1,86 GHz @ 2,54 GHz
Operační paměť	A-Data Viesta Extrema 2x2GB DC DDR2 800MHz CL4
Grafická karta	Asus Geforce 7600 GT 256 MB PCI-E
Operační systém	Microsoft Windows Vista Bussiness SP1 x64
Verze ovladačů grafické karty	Driver Instrumentation 7.15.11.6375

Verze ovladačů grafické karty, která byla použita, se dodává s aplikací PerfHUD od společnosti NVidia, jedná se o developer verzi driverů, která umožňuje low level profiling grafické karty. Bohužel díky těmto možnostem drivery ovlivňují výkon strátou 6-7% výkonu.



Testovací scéna s ohněm

Testovací scéna bez ohně

Hra byla testována ve fullscreenu při rozlišení 1680x1050 na scéně, která je vidět na obrázku viz výše. Scéna byla testována:

Scéna	Počet snímků za vteřinu (FPS)	snížení výkonu v %
Bez ohně (levý screenshot)	60	0,00%
S ohněm blíže pororovateli	30	50,0%
S ohněm v ruce (pravý screenshot)	44	26,6%
S ohněm (u domu)	54	10,0%
Se dvěma ohni (u domu)	51	15,0%
Se třemi ohni (u domu)	48	20,00%

Dále bylo ještě testováno na již starším notebooku (bylo použito stejné LCD se stejným rozlišením a fullscreen) s parametry:

Chipset	ATI Xpress 200M
Procesor	AMD Sempron 3100+ 1,8 GHz
Operační paměť	1,5 GB DDR 333 MHz CL3
Grafická karta	ATI X300 se sdílenou pamětí
Operační systém	Microsoft Windows Vista Bussiness SP1
Verze ovladačů grafické karty	ATI Catalyst 8.4

Scéna	Počet snímků za vteřinu (FPS)	snížení výkonu v %
Bez ohně (levý screenshot)	20	0,00%
S ohněm blíže pororovateli	2	95,0%
S ohněm v ruce (pravý screenshot)	3	85,0%
S ohněm (u domu)	12	40,0%
Se dvěma ohni (u domu)	9	55,0%
Se třemi ohni (u domu)	7	65,00%

Testy bylo zjištěno, že výkon je ovlivněn především při renderingu ohně a je nutno zapnout alpha blending. Při renderování není možné předběžně vyhodnotit z-buffer a rozhodnout, které pixely budou vidět a které ne. Proto musíme pro všechny pixeli, počítat všechny výpočty.

Dále si lze v testech všimnout, že výkon je ovlivněn nejvíc, pokud je oheň blízko pozorovateli. To je způsobeno tím, že čím blíže je oheň pozorovateli, tím více se renderuje pixelů z ohně a tím více se provádí výpočtů pro jednotlivé pixely.

Dále je výkon také ovlivněn díky aktivovanému alpha blendingu, protože se musí provádět jak zápis tak čtení dat z/do videopaměti, místo pouze čtení. Což má za následek větší datový tok mezi videopamětí a GPU (dvojnásobný).

Z důvodu použité techniky renderování jsou pro spuštění hry velmi nevhodné grafické karty bez videopaměti, tzv. grafické karty se sdílenou pamětí. U těchto grafický karet se přistupuje do systémové paměti, čímž se celý rendering ještě přibližně 5 krát zpomalí.

5. Závěr

V bakalářské práci jsme si řekli několik technik, které se dají použít pro generování ohně a byly řečeny též jejich výhody a nevýhody. Dále jsme si ukázali, jak naimplementovat oheň včetně světla, které vrhá, a který je vhodný pro použití do herního enginu a je relativně jednoduše modifikovatelný. Dále v práci byly ukázány optimalizace, které se dají použít pro zvýšení počtu snímků za vteřinu a testy výkonu, které naznačily, jak je vhodné oheň používat.

Co se týče porovnání s původními představami, tak přesto, že se oheň podobá ohni, je na něm stále co vylepšovat. Vhodná vylepšení by mohly být úprava textury ohně, jiná funkce pro výpočet alpha hodnoty u jednotlivých částic, použití několika emitorů pro vystřelování částic nebo rozšíření ohně na volumetrický oheň (na oheň není nutné koukat pouze z jedné strany, obsahuje i „hloubku“, mohlo by jít např. postavením dvou ohňů naproti sobě a udělat mezi částicemi interpolaci). Přesto, že oheň je již dosti optimalizovaný, tak by bylo vhodné provést ještě další optimalizace (např. LOD efektu ohně, popř. optimalizace pixel shader části) protože momentálně oheň (včetně světel od ohně) tvoří většinu zatížení herního enginu (přesné zatížení je možné vyčíst z tabulky z kapitoly „měření výkonu“ – maximální naměřené zatížení bylo 50% a minimální 10%).

Literatura

AMD Corporation. (1. January 2008). *Developer*. (AMD Corporation) Získáno 1. April 2008, z Developer: http://ati.amd.com/developer/dx9/ATI-DX9_Optimization.pdf

GameDev.net. (1. January 1999). *Gamedev.net*. (Gamedev.net) Získáno 30. April 2008, z Forum: http://www.gamedev.net

Gamedev.net. (1. January 2008). *Gamedev.net*. (Gamedev.net) Získáno 30. April 2008, z Articles and Resources: http://www.gamedev.net/reference/

Institute of Data Analysis and Visualization (IDAV) and Department of Computer Science,. (2006). *http://www.cs.ucdavis.edu/*. Získáno 30. April 2008, z http://www.cs.ucdavis.edu/: http://www.cs.ucdavis.edu/research/tech-reports/2006/CSE-2006-22.pdf

Microsoft Corporation. (2008, May 30). *MSDN*. (Microsoft Corporation) Retrieved April 30, 2008, from MSDN Library: http://msdn.microsoft.com/Library

Microsoft Corporation. (1. January 2007). *XNA Creators Club Online*. (Microsoft Corporation) Získáno 30. April 2008, z Education: http://creators.xna.com/Education/

NVidia Corporation. (1. January 2008). *Developer Zone*. (NVidia Corporation) Získáno 30. April 2008, z DirectX: http://developer.nvidia.com/page/directx.html

NVidia Corporation. (2004). *GPU Gems.* Boston: Addison-Wesley.