

Animace ohňostroje

(Projekt do předmětu MSI - Modelování a simulace)

Obsah

Obsah	1
Zadání	2
Animace ohňostroje	2
Informace o projektu do MSI	2
Řešení projektu	3
Programové prostředí	3
Další použité nástroje	4
ImageMagick	4
MEncoder	4
Modelování	5
Diskrétní prvky	5
Odpalovací lanko	5
Uzel na lanku	5
Spojité prvky	6
Svítící objekt	7
Raketa	7
Simulace	9
Diskrétní prvky	9
Spojité prvky	11
Vizualizace	12
Ukázkový příklad	13
Nápad	13
Implementace	13
Odpalovací mechanismus	14
Výsledná animace	15
Závěr	16
Použitá literatura	17

Zadání

Animace ohňostroje

Modelujte ohňostroj jako systém částic pro které platí fyzikální zákony, zejména gravitace, hmotnost, hybnost atd. Výstupem simulace bude vizualizace stavu v různých časových okamžicích - tj. jednotlivé snímky, ze kterých bude možno následně sestavit ukázkové video.

Informace o projektu do MSI

Hlavním cílem projektu je vyhledání a simulační zpracování vlastního zadání. Projekt bude hodnocen až dvaceti body a musí mít některé náležitosti:

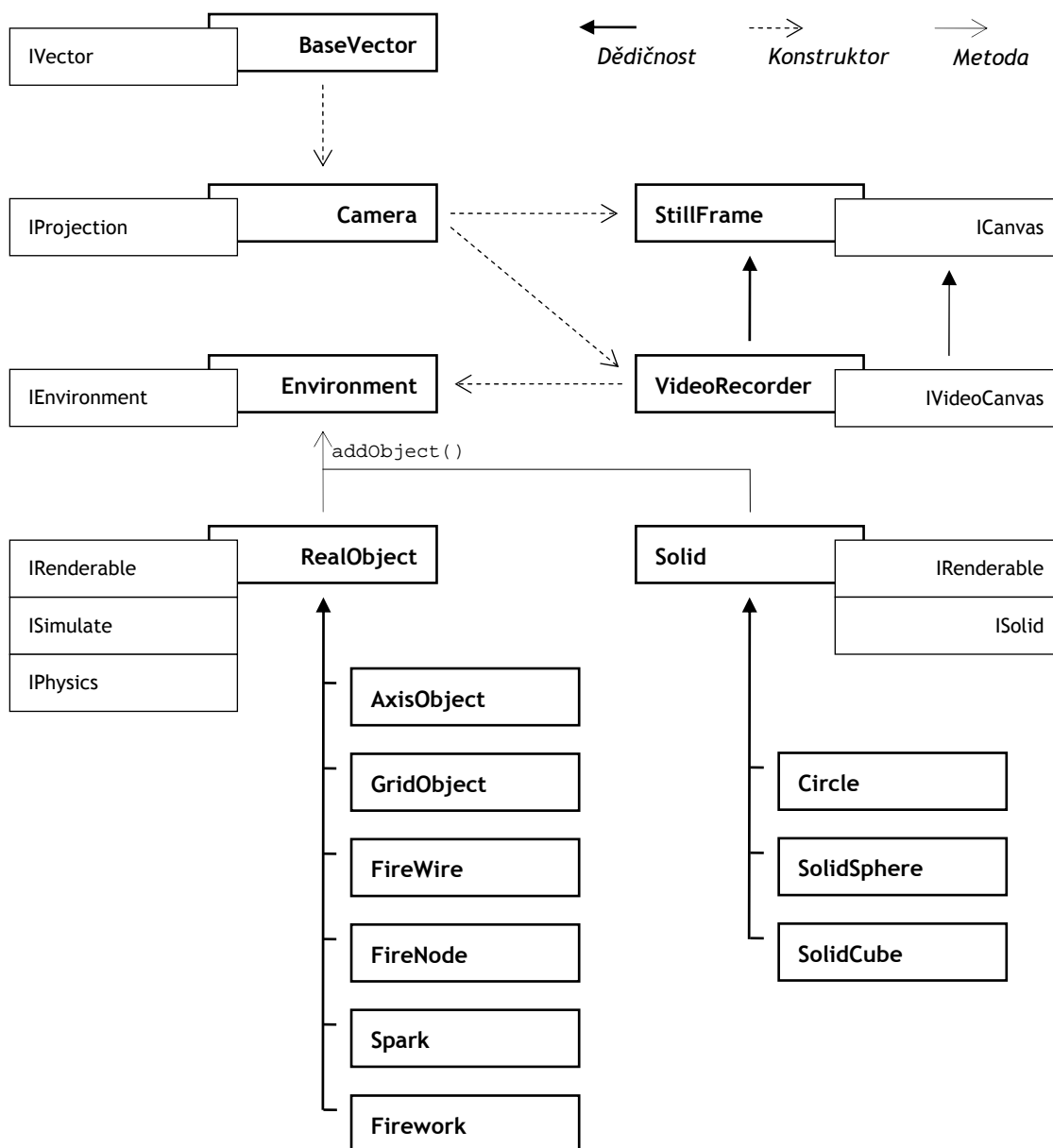
- formulace zadání problému - hodnotí se náročnost zadaného problému. Příliš jednoduché projekty budou ztrácet až polovinu bodů. Máte-li pochybnost o úrovni vašeho zadání, napište mi pro kontrolu své zadání. Za dostatečně složitý problém se považuje diskrétní systém s nejméně dvěma typy procesů a spojitý systém s nejméně dvěma rovnicemi. Vzhledem ke zkušenostem z minulých let se zcela zapovídá zadání typu supermarket, menza a restaurace (posledně jich bylo asi 60%) - tyto projekty nebudou uznány.
- řešení projektu má odrážet získané znalosti v kurzu MSI - projekt volte jako diskrétní, spojitý, případně kombinovaný systém. Klasickým diskrétním problémem je systém hromadné obsluhy (z pochopitelných důvodů se zcela zapovídá použití nějakého zadání z přednášek). Pro simulaci použijte například knihovnu SIMLIB (vítá se i použití jiného simulačního prostředku). Pokud by někdo nechtěl řešit klasický problém jako z přednášky, bude vítán. Jako alternativní projekt může být implementace nějaké simulační metody, modelování znalostí, modelování inteligence, modelování agentů. Alternativní a mimořádně kvalitní projekty mohou být hodnoceny s přidáním bonusových bodů!
- budu klást důraz na kompletnost zpracování - zadání, analýza problému, abstraktní model, simulační model, ověření správnosti modelu (abstraktního, simulačního), provedení série experimentů s různými parametry modelu, vyhodnocení výsledků, komentář výsledků a návrh na optimalizaci (poučení se z experimentu. Je-li projektem například SHO, pak návrh na optimalizaci systému a podobně).
- termín odevzdání projektu stanovuji na 7. leden 2005 do 15 hodin. Po tomto termínu přijmu projekt pouze ve zvláštním případě (onemocnění v období několika dnů před termínem NENÍ důvodem pozdějšího odevzdání). Projekty odevzdané ještě v prosinci budou hodnoceny lépe.
- projekt odevzdávejte v elektronické formě přes IS FIT (časem tam vytvořím příslušný termín). Své soubory zabalte do formátu .tar.gz (.tgz) a pojmenujte svým login jménem (př. xnovak99.tgz). Archiv by měl obsahovat všechny podstatné části - texty (pdf, ps, obyč. txt - rozhodně NE MS-Word!!!), programy, simulační výstupy. Pokud nebudu schopen váš projekt prohlížet na svém linuxovém počítači, projeví se to v hodnocení.
- připomínám, že odevzdání projektu (a jeho hodnocení alespoň pěti body) je nutnou podmínkou pro udělení zápočtu. Bez zápočtu nejste připuštěni k závěrečné zkoušce.

Inspirace: modely hardwaru, komunikační protokoly, doprava, služby, výrobní procesy, fyzikální jevy a zákony, elektro (např. obvody). Aplikace metody Monte-Carlo, analytické modely. Ocením každou aktivitu vedoucí k poznání nových věcí v modelování a simulaci (nové nástroje, jazyky, metody...).

Řešení projektu

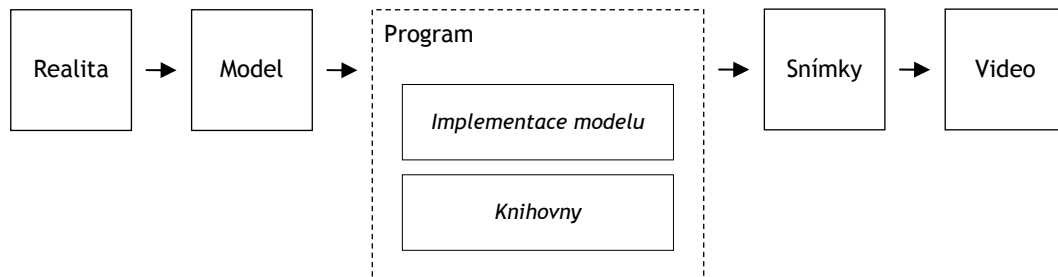
Programové prostředí

Pro řešení tohoto projektu jsem použil skriptovací programovací jazyk PHP. Tento jazyk je sice určen primárně pro psaní skriptů pro web, ale v nové verzi (od 5.0.0) obsahuje taky prostředky které ulehčují jeho všeobecné použití - zejména jde o plnou podporu objektově orientovaného programování (třídy, rozhraní, dědičnost, viditelnost proměnných a metod, abstraktní a statické metody). V implementaci jsem se snažil využít těchto možností co nejvíc a navrhnout různé třídy a rozhraní (obr. 1). Některé části projektu jsou odděleny pro potřeby tohoto konkrétního projektu možná i zbytečně - většinou z obavy že u dalších úprav bude stejně nutnost kód rozdělit.



Obr. 1: Diagram tříd

Jelikož jde o modelování a simulaci ohňostroje, tj. vizuálního efektu, jako nejvhodnější výstup simulace se jeví animace daného děje. Animace v tomto projektu je vytvořena z jednotlivých snímků které dostáváme vizualizací stavu prostředí (obr. 2). Z těchto snímků je pak možno sestavit buď animovaný obrázek (GIF) nebo video (AVI). První varianta je omezena na 256 barev, ale používá bezztrátovou kompresi, takže jsem ji používal přednostně. S videem byl větší problém - žádný způsob komprimace nevyhovoval kvalitativně a nekomprimované video mělo až tak velký datový tok, že ho nebylo možno přehrávat.



Obr. 2: Vytvoření animace

Další použité nástroje

ImageMagick

<http://www.imagemagick.org/>

ImageMagick je kolekce programů pro práci s obrázky, především konverzi mezi formáty a aplikace filtrů. Pro tento projekt jsem využil z tohoto balíku jenom programu `convert`, kterým jsem vytvářel animace následovným způsobem:

```
convert -delay 4 ./test/*.png ./test.gif
```

Parametrem `-delay 4` se udává délka trvání snímku v setinách sekundy, takže výslední animace v tomto případě bude mít snímkovací frekvenci 25 Hz. Program `convert` umožňuje i vytvoření animace typu MNG, ale nenašel jsem jednoduchý způsob prohlížení.

Problémy se kterými jsem se setkal při realizaci projektu ohledně tohoto balíku byli:

- nemožnost vytvoření animace když se použilo bílé pozadí
- chybné vytvoření animace kde se nacházelo více barev
- velká náročnost na paměť a z toho vyplývající pomalost při používání odkládacího souboru (swap) u animacích delších nebo pokud byly ve větším rozlišení

První dva problémy jsem vyřešil za pomoci programu Animation Shop 2 z balíku Paint Shop Pro verze 6 (shareware, jen pro operační systém MS Windows) a kvůli třetímu jsem si musel odpustit animace ve velkém rozlišení.

MEncoder

<http://www.mplayerhq.hu/>

MEncoder je součástí projektu MPlayer a slouží pro vytváření komprimovaných videí. V projektu jsem ho chtěl využít na vytvoření videa s vysokým rozlišením ale kvalita výstupu byla mizerná - obraz byl plný artefaktů (konkrétně šlo o 1280x800 obrazových bodů, rozlišení nativní pro můj monitor). Dané chování se dá vysvětlit - použitá komprese (MPEG4) je určena pro komprimování realistických obrazů (fotky, film) a na komprimaci obrazů s vysokým kontrastem (síťový model v našem případě) je nevhodná. Pro úplnost byl příkaz pro vytvoření videa následovný:

```
mencoder mf://./test/*.png -mf type=png:fps=25 \
  -ovc lavc -lavcopts vcodec=mpeg4:vbitrate=6000 \
  -oac copy
-o test.avi
```

Zde je taky vidět že video má 25 snímků za vteřinu, je komprimováno pomocí ISO MPEG4 s proměnlivým datovým tokem 6000 kilobitů za vteřinu (což je relativně hodně pro video v MPEG4).

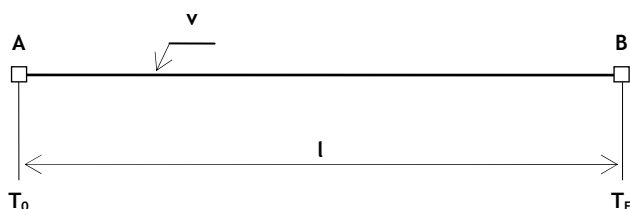
Modelování

Tato kapitola popisuje způsoby modelování jednotlivých objektů použitých při ohňostroji. Tyto prvky lze rozdělit do dvou kategorií - diskrétní a spojité. Mezi diskrétní patří třeba odpalovací ústrojí sestávající z odpalovacích lanek které mohou být na některých místech svázané uzly a slouží k časování následovních akcí. Tyto akce jsou už pak spojitým systémem, kde je raketa jako objekt poháněn raketovým motorem a působí na ni gravitace a odpor vzduchu. Po spotřebování paliva nastává exploze (zde je to modelováno jako diskrétní prvek, ve skutečnosti jde o spojitý jev) u které se uvolní určitá energie a její část se použije na udání výchozí rychlosti jednotlivých úlomků. Na tyto úlomky se znova vztahují fyzikální jevy jako gravitace a odpor vzduchu a jsou modelovány spojitě.

Diskrétní prvky

Odpalovací lanke

Lanke je definováno dvěma body A, B (reprezentovány 3D vektory) označujícími jeho konce (resp. počátek a konec) a skalární hodnotou v která udává rychlost šíření ohně po lanke v jednotkách vzdálenosti na jednotku času (obr. 3). Pokud jsou polohy udány v metrech, tak rychlost by měla mít rozměr metry za sekundu.



Obr. 3: Model odpalovacího lanka

Diskrétní vlastností lanka je schopnost spustit další proces v čase T_E , tj. když oheň dosáhne bodu B. Tento čas se určuje z jeho skutečné délky (vzdálenost dvou bodů v prostoru, vztah 2) a vlastností v , pomocí vztahu 1:

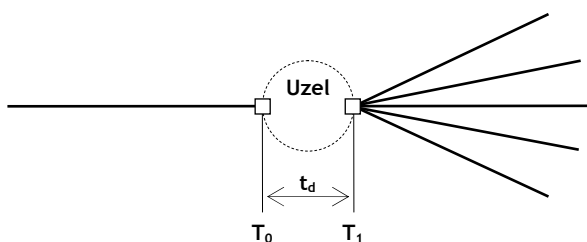
$$T_E = \frac{l}{v} = \frac{|AB|}{v} \quad (1)$$

$$|AB| = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2} \quad (2)$$

I když je lanke samo o sobě diskrétní prvek, simuluje se spojitě jako ostatní součásti modelu ohňostroje - průběžně se počítá poloha ohně a když dosáhne konce tak lanke „umírá“ a pokud existuje (je přiřazen) následovník tak je aktivován. Protože následovníkem může být jenom jeden objekt, tak byla nutnost implementovat uzly popsané dále.

Uzel na lanke

Objekt uzel je charakterizován vlastností zpoždění t_d . Toto zpoždění udává délku života uzlu a taky znamená časový rozdíl mezi aktivací následovníků uzlu a uzlu samotného. Uzel je většinou aktivován pomocí lanka (které ho má jako jediného následovníka) a po uplynutí doby zpoždění jsou aktivovány následovníci kterých může být už libovolný počet. Zjednodušení uzlu oproti realitě je, že se nepočítá s náhodnou aktivací následovníků - následovníci jsou aktivovány ve stejný okamžik.



Obr. 4: Model uzlu

Spojité prvky

Spojité modely jsou modely fyzických těles, v našem případě jde o raketu a to co z ní zbude po explozi. Na tyto objekty se vztahují fyzikální jevy a jejich zákony - v projektu uvažují o gravitaci, odporu vzduchu a v případě raket taky o nějakém pohonu. Společným prvkem všechny těchto jevů je, že je lze modelovat přes působení sil.

Základním prvkem realistického modelu je gravitace. Gravitační sílu modelují jako konstantní sílu působící směrem dolů (v souřadné soustavě je to záporná hodnota souřadnice z) podle vztahu 3:

$$\vec{F}_g = m \cdot \vec{g} \quad (3)$$

Gravitace je teda v našem projektu zjednodušena prakticky na jeden vektor vynásobený hmotností objektu, ale pokud bychom chtěli být důslední, měli by jsme ji modelovat jako vektor působící do středu země (tj. koule o poloměru asi 6372 km) anebo za pomoci vztahu pro gravitační sílu mezi dvěma objekty. Pro náš projekt postačí ale zjednodušená gravitace implementována následovně:

```
if ( $this->mass > 0 ) { // physics of gravitation
    $m = $this->mass; // [kg]
    $g = new BaseVector( 0, 0, -9.8 ); // [m / s^2]
    $Fg = $g->getScaled( $m ); // Fg = m.g [ N = kg.m/s^2 ]
    $forces['Fg'] = $Fg;
}
```

Dalším všeobecně působícím jevem je odpor prostředí, u ohňostroje je to konkrétně odpor vzduchu. Tento jev jsem musel zakomponovat pro realističnost výbuchu protože první simulace exploze, kde se s odporem nepočítalo vypadala na první pohled jako neskutečná - něco tomu chybělo. Efekt který jsem postrádal v té animaci byl ten, že při výbuchu částice letí poměrně velkou rychlostí a zpomalují se zvětšujícím se průměrem (resp. časem). Po chvíli uvažování jsem přišel na nápad implementovat odpor prostředí. Vztah 4 jsem našel na jednom internetovém fóru (zdroj uveden v literatuře) ale pro naše potřeby je příliš složitý, takže jsem ho zjednodušil spojením konstant na tvar 5:

$$F = \frac{1}{2} \cdot C \cdot \sigma \cdot S \cdot v^2 \quad (4)$$

kde:

- F odporová síla
- C součinitel odporu, závisí na tvaru tělesa
- σ hustota prostředí
- S průřez tělesa ve směru rychlosti
- v rychlost tělesa

$$\vec{F}_r = -\frac{\vec{v}}{|\vec{v}|} \cdot C \cdot |\vec{v}|^2 = -\vec{v} \cdot C \cdot |\vec{v}| \quad (5)$$

Protože ve svých modelech uvažují jen o hmotných bodech, které nemají reálně zadané rozměry ale jenom hmotnost, konstanta C reprezentuje jak součinitel odporu, tak i tvar a velikost (průřez) tělesa. Toto zjednodušení je přijatelné pro tento projekt, protože účelem je dosáhnout realistického efektu exploze a její animace, ne vědecky přesných výpočtů. Pro úplnost je zde úryvek kódu implementující odpor prostředí:

```
if ( ($v = $this->velocity->getLength()) > 0 ) {
    $forces['Fr'] = $this->velocity->getScaled( - $this->resistance * $v );
}
```

Svítilící objekt

Svítilícím objektem jsem nazval objekt - úlomek rakety vzniklý po explozi. Tento objekt nemá žádný pohon, jen se mu při vzniku přiřadí počáteční rychlost. Toto přiřazení odporuje teorii o spojitém modelování protože správně by se mělo počítat s působením síly vzniklé v důsledku exploze, takže to je další zjednodušení analytického modelu na simulační.

Dvě vlastnosti které má svítící objekt navíc oproti základnímu objektu je doba života a relativní velikost. Tyto vlastnosti neslouží pro modelování fyzikálních dějů ale jenom pro vizualizaci, kde se objekt kreslí jako vektory o stejné délce (relativní velikost) a náhodné orientaci. Velikost těchto vektorů se zmenšuje lineárně s časem a po uplynutí doby života objekt zaniká a není ho potřeba dále simulovat protože by stejně nebyl vidět.

Objekt taky zaniká když jeho výška klesne pod nulu, což se dá chápat jako modelování nad vodní hladinou. Alternativně by se mohl modelovat dopad na měkký povrch (písek), kde by byla energie dopadu pohlcena a objekt by jednoduše ztratil svou hybnost (takže prakticky by stačilo rychlost vynulovat) a dosvítit by si tam kde padnul. Další alternativa by mohla spočívat v modelování odrazu od tvrdého povrchu - část energie by byla pohlcena ale další část zachována a objekt by byl teda odražen - při dokonalém odrazu jde o zrcadlení vektoru hybnosti (tj. prakticky jen rychlosti).

Raketa

Raketa jako základ ohňostroje je modelována nejprve za pomoci vzorce pro raketový motor (vztah 6). Parametry rakety které je nutno znát jsou pozice, směr natočení, hmotnost, procentuální poměr paliva vzhledem k celkové hmotnosti, spotřeba paliva za jednotku času a rychlost unikajících zplodin. Z těchto parametrů se pak dá namodelovat jak pohon, tak diskrétní prvek - zpoždění exploze která nastane po spotřebování paliva.

Vztah pro raketový pohon je následovný (získáno v rámci rozhovoru na IRC se studentem 1. ročníku fyziky na Masarykové univerzitě):

$$M \cdot \vec{a} = R \cdot \vec{u} \quad (6)$$

kde:

M	hmotnost rakety	kg
a	dosažené zrychlení	$\text{m} \cdot \text{s}^{-2}$
R	spotřeba paliva	$\text{kg} \cdot \text{s}^{-1}$
u	rychlost unikajících zplodin	$\text{m} \cdot \text{s}^{-1}$

Ve svém projektu jsem potřeboval znát jen okamžitou sílu kterou vyvine motor v daném čase, tj. tah motoru a to je přesně to co je na pravé straně vztahu 6. Protože první simulace kde byl úbytek paliva konstantní se mi znova nezdála skutečná, zavedl jsem proměnlivý úbytek paliva, pro jednoduchost lineárně klesající z 1,8 násobku průměrné hodnoty na hodnotu takovou, aby se palivo spotřebovalo za čas který by se dal vypočítat z průměrného úbytku. Tento způsob výpočtu modeluje jednoduchý pokles tahu motoru, který by se pořádně měl modelovat nejspíš přes křivky účinnosti dodávky paliva do motoru a křivku efektivity motoru.

Znova teda jde o zjednodušení analytického modelu na simulační a ten je pak implementován následovným kódem, ve kterém je vidět využití dědičnosti v objektově orientovaném návrhu - metoda `getForces()` nejprve zjistí síly působící na obyčejný/základní objekt a přidá k nim tah motoru:

```
function getForces() {
    $forces = parent::getForces();
    $forces['Fm'] = $this->fuelSpeed->getScaled( $this->getCurrentFuelDec() );
    return $forces;
}

private function getCurrentFuelDec() {
    $Te = $this->fuelMax / $this->fuelDecreasePerSecond;
    $r = $this->time / $Te;
    $max = 1.80;
    $min = 2-$max;
    return $this->fuelDecreasePerSecond * ( $max - $r * ( $max - $min ) );
}
```

Další část modelu rakety je model její exploze. U exploze se uvolňuje energie E (její hodnotu dostaneme z hmotnosti rakety a koeficientu), ze které se část promění na světlo, část se na teplo a co zbude bude použito na urychlení částic - kinetická energie:

$$E = E_{light} + E_{therm} + E_k \quad (7)$$

Předpokládáme že se energie rozdělí rovnoměrně na všechny úlomky:

$$E_{1k} = \frac{E_k}{N} \quad (8)$$

Tuto kinetickou energii pak můžeme použít k výpočtu rychlosti úlomků:

$$E_{1k} = \frac{m \cdot v^2}{2} \quad (9)$$

$$v = \sqrt{\frac{2 \cdot E_{1k}}{m}} \quad (10)$$

Hodnota ze vztahu 10 udává jen velikost rychlosti, směr rychlosti je pro každý úlomek jiný a mělo by platit že se úlomky rozprostřou rovnoměrně po ploše koule. Toto nelze jednoduše naimplementovat, takže v simulačním modelu jsou směry vybírány náhodně a při větším počtu částic se předpokládá že budou poměrně rovnoměrně.

Dále pak platí zákon zachování hybnosti:

$$\vec{p} = \sum_{i=1}^N \vec{p}_i \quad (11)$$

$$m \cdot \vec{v} = \sum_{i=1}^N m_i \cdot \vec{v}_i \quad (12)$$

Tento zákon má dva důsledky. Prvním, výraznějším, je to, že pokud měla raketa před explozí nějakou rychlost, tak tato rychlost se přičte ke všem úlomkům, tj. po explozi se bude „ohnivá koule“ pohybovat zpočátku ve směru v jakém letěla raketa. Druhým důsledkem je, že exploze bude symetrická podle přímky ve směru pohybu rakety pokud se počítá s úlomky stejné velikosti.

Implementován je jenom první důsledek zákonu zachování hybnosti a to tak, že se k náhodné rychlosti úlomku připočte vektor rychlosti rakety (v době exploze). Druhý efekt není nutný, protože má nepatrný vliv na vizualizaci. Implementace by ale nebyla složitá - stačilo by vygenerovat proti-úlomek s otočenou náhodnou rychlostí - matematicky by pak součet hybností vyšel správně.

Kód pro modelování exploze je teda v konečné podobě zapsán následovně:

```
$spark = array();
$N = round( $this->mass / $this->childMass ); // approx. number of childs
$E = $this->mass*$this->energyPerKg; // total energy
$Ek = 0.9 * $E; // - 10% therm+light
$mass = $this->mass / $N; // exact child mass
$v = sqrt( 2*$Ek / $mass ); // initial velocity size
for($i=0;$i<$N;$i++) {
    $spark[$i] = new Spark( $this->position, $mass, 1
        , $this->childLives*rand(80,120)/100 );
    $V = new BaseVector( rand(0,200)-100, rand(0,200)-100, rand(0,200)-100 );
    $spark[$i]->velocity = $V->getNormalized()->getScaled( $v )
        ->add( $this->velocity );
    $spark[$i]->resistance = 1/10/$N/$N; // S == A.x^2 => /N/N
    $spark[$i]->color = clone $this->color;
    $this->ENV->addObject( $spark[$i] );
    $this->ENV->activate ( $spark[$i], $runtime );
}
```

Simulace

Pokud máme vytvořen model, můžeme přistoupit k dalšímu kroku - simulaci v čase. V projektu jsou implementovány dva algoritmy - jeden pro poměrně přesné určení času kdy nastává diskrétní jev a druhý pro spojitou simulaci fyzikálních jevů pomocí působení sil. Objekt, který lze obecně simulovat, musí implementovat následující rozhraní obsahující metody s jednoduchým významem - je-li objekt aktivní, aktivace objektu, simulace a provedení vazby při úmrtí objektu:

```
interface ISimulate {  
    function isActive ();  
    function activate ();  
    function simulate ( $dtime );  
    function acceptDeath( $rtime );  
}
```

Pokud na objekt mají vplyv fyzikální jevy jako gravitace a odpor vzduchu, tak musí implementovat taky toto rozhraní, které obsahuje metodu na vyčíslení působících sil:

```
interface IPhysics {  
    function getForces();  
}
```

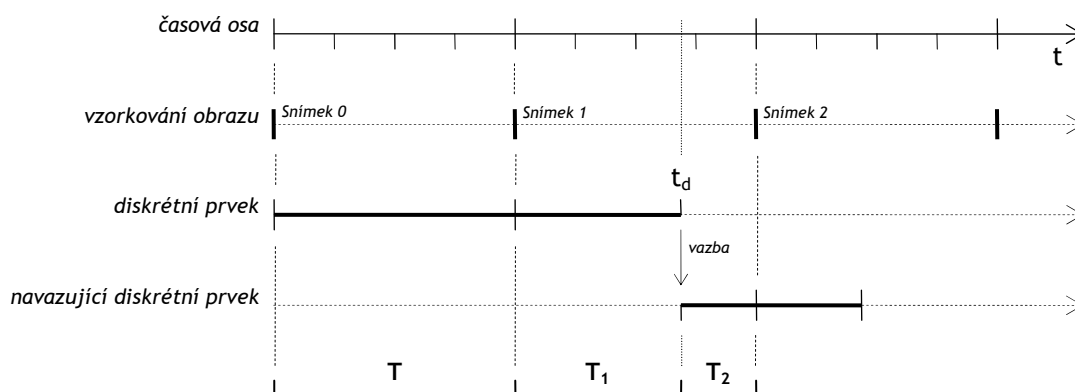
Obě rozhraní jsou implementovány již v základním objektu a pro složitější (odvozené) objekty se nabízí možnost zřetězení volání (např. vyčíslení základních sil a přidání tahu motoru).

Diskrétní prvky

Diskrétní prvky jsou paradoxně simulovány taky spojitě, alespoň v mém projektu je to tak. Základním principem je opakovaná simulace v krocích, kterých délka se odvozuje od vzorkovací frekvence obrazu (FPS) dle vztahu 13:

$$T = \frac{1}{FPS} \quad (13)$$

Fakt, že je simulační interval T odvozen právě ze vzorkovací frekvence obrazu souvisí s tím, že nás zajímá pouze situace v modelu v čase příslušejícím vytvoření snímku. Protože není vše tak jednoduché jak to vypadá, i zde je jeden obrovský problém - a to ten, že co s tím když prvek končí v čase mezi 2 snímky (situaci ilustruje obr. 5).

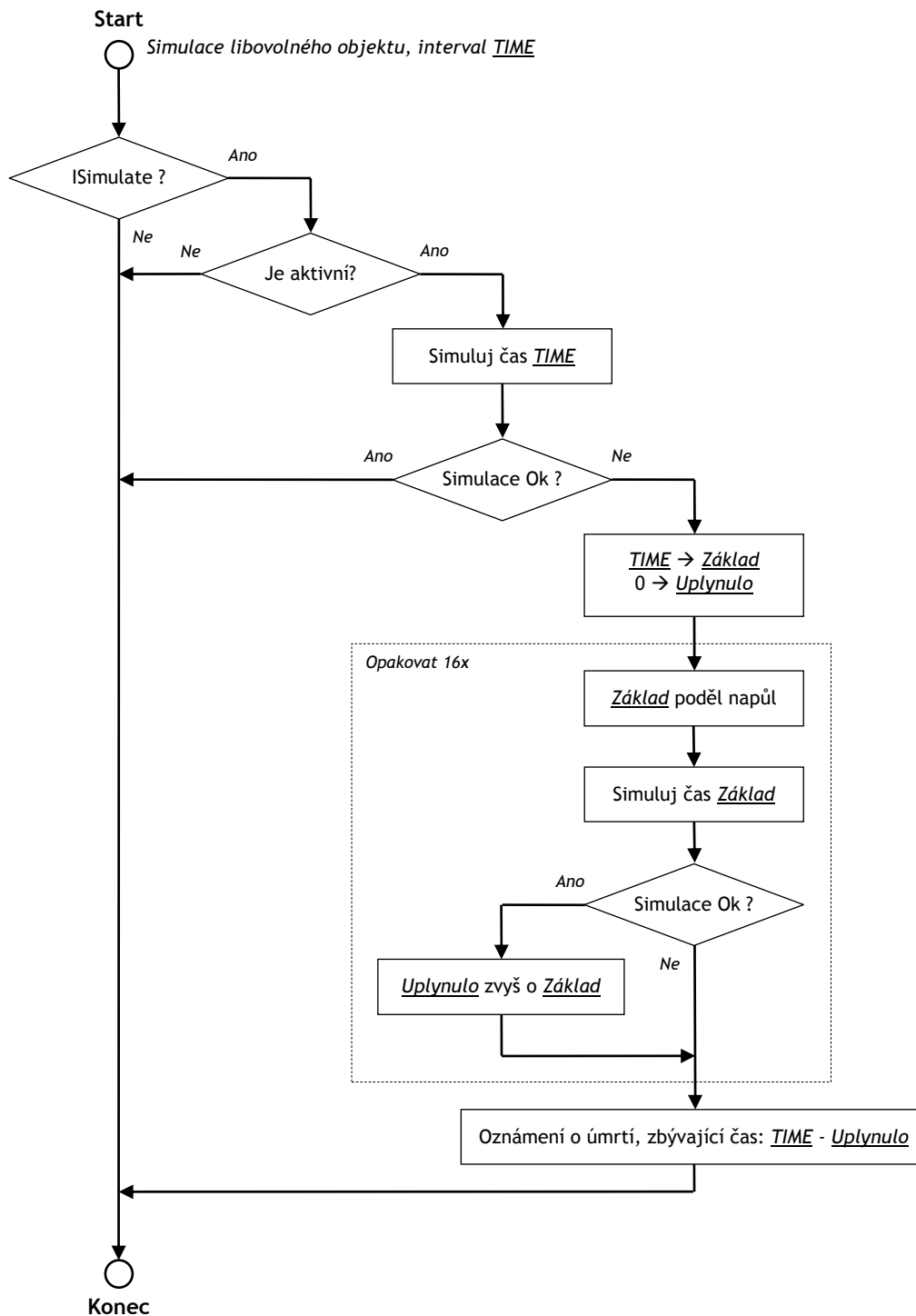


Obr. 5: Simulace diskrétních prvků

Řešení tohoto problému spočívá v určení přesné doby života, resp. času „úmrtí“ prvku (t_d). Na určení doby úmrtí jsem vymyslel iterativní algoritmus, který využívá stávající metodu sloužící k simulaci. Pravidla pro simulační metodu jsou následující:

- vstupem je časový interval dt
- pokud prvek neumře, aktualizuj stav na stav v čase „ $t_{ed}+dt$ “
- pokud prvek umře v době dt , stav se nemění a metoda indikuje, že simulaci nelze provést pro zadaný interval

Pokud objekty vyhovují této specifikaci, lze použít algoritmus založený na půlení intervalu (obr. 6) který vypočte přibližnou dobu t_d pomocí iterací.



Obr. 6: Algoritmus na určení doby úmrtí

Souvislost intervalů z časového diagramu na obrazu 5 s algoritmem:

T	základní interval	<u>TIME</u>
T_1	čas od počátečního bodu do doby t_d	<u>Uplynulo</u>
T_2	čas od t_d k dalšímu bodu	<u>TIME - Uplynulo</u>

Doba T_2 je vlastně parametrem metody `acceptDeath($ptime)`. Tato metoda slouží k aktivaci následníků kteří se ihned po aktivaci simulují s časem $TIME=T_2$ tak aby dosáhli stavu po aktuálním simulačním kroku.

Spojité prvky

Simulace spojitých prvků v tomto projektu spočívá v simulaci fyzikálních jevů. Všechny jevy se kterými počítáme jsem zjednodušil na působení sil. Simulace probíhá zas v krocích (jak bylo popsáno výše) a v každém kroku se provede vyčíslení působících sil, jejich sečtení a pokud se právě nevyruší tak zpětná aplikace společně působící síly na objekt. Kód zajišťující tuto zpětnou aplikaci je následovně:

```
function simulate( $dtime ) {  
  
    // update local time  
    if ($this->active) $this->time += $dtime;  
  
    // reality simulation?  
    if ($this instanceof IPhysics)  
    if ($forces = $this->getForces()) {  
        $F = BaseVector::getZero();  
        foreach($forces as $Fi) $F = $F->add( $Fi );  
        if (($F->getLength()>0)&&($this->mass>0)) { // there will be some effect  
            // computations  
            $acceleration = $F->getScaled( 1/$this->mass ); // F=m.a -> a=F/m  
            $dv = $acceleration->getScaled( $dtime ); // dv = a.dt  
            $ds = $this->velocity->getScaled( $dtime ); // ds = v.dt  
            // change of state  
            $this->position = $this->position->add( $ds );  
            $this->velocity = $this->velocity->add( $dv );  
        }  
    }  
  
    return true; // simulation successful  
}
```

Tento algoritmus aktualizuje stav objektu, tj. pozici a rychlost, na takový, který by byl po uběhnutí časového intervalu $\$dtime$. Tento interval by měl být volen co nejkratší aby se zaručila přesnost, protože uvedený způsob výpočtu je ve své podstatě numerický výpočet těchto integrálů:

$$\vec{v} = \int \vec{a} dt \quad (14)$$

$$\vec{s} = \int \vec{v} dt \quad (15)$$

V prvních simulacích, kde se ještě nepočítalo s odporem prostředí, jsem volil simulační interval dt stejně dlouhý jako byla perioda vzorkování (vztah 13). Toto nastavení fungovalo. Po přidání odporu prostředí se začali množit chybně generované obrazy - objekty které měli velikou rychlost zmizely z obrazu. Nejprve jsem neměl tušení, že čím to bude ale pak se mi povedlo vygenerovat dva za sebou následující snímky ze kterých už bylo jasno o co jde - na druhém obrazu byly úlomky po explozi blíže k sobě než na první.

Vysvětlení tohoto jevu je následující: odporová síla je kvadrátem rychlosti, tudíž při veliké počáteční rychlosti je síla která působí v protisměru extrémně velká. Tato síla v součinnosti s poměrně velkým dt (bylo to 40 ms, odvozeno z 25 snímků za vteřinu) vyvolala zrychlení převyšující aktuální zrychlení objektu a tím vlastně otočila jeho směr.

Problém jde řešit více způsoby:

- nepoužívat dt pro numerický výpočet integrálu ale integrály vyřešit analyticky a pak bude simulace fungovat s libovolným intervalem
- výpočet provést s jemnějším krokem vícekrát

U výpočtu doby úmrtí objektu jsem zvolil iterační přístup namísto analytického řešení a tady tomu nebylo jinak. Problém jsem tedy vyřešil poměrně rychle za pomoci hrubé síly - podělením intervalu mezi snímky na několik částí (negativním jevem je větší časová náročnost):

```
$dt = 1 / $FPS / $PRECISION;  
for( $p=0; $p<$PRECISION; $p++ ) $universe->runSimulation($dt);
```

Vizualizace

Vizualizace je další důležitou částí projektu, protože nám umožňuje jednodušeji interpretovat výsledky, v našem případě výsledky simulace. Z jednotlivých snímků pak dostáváme animace a pomocí ní můžeme upravovat model - přidávat další detaily které jsme při prvním pokusu vynechali v zápalu zjednodušování.

Nejlepší vizualizace je vždy ta, která je nejvíc reálná, takže jsem rovnou zvolil 3D prostředí, které se transformuje na 2D obraz pomocí složitých matic v pohledové transformaci. Vše co jsem potřeboval znát k napsání zobrazování jsem získal z absolvovaných předmětů (především jde o Základy počítačové grafiky) a dále pak z Internetu (konkrétní výpočet zobrazovací matice a matic pro operace zvětšení, posunutí a otočení).

Podrobný popis vizualizace tady není z důvodu obsáhlosti problematiky, ale pro ilustraci je zde vizualizace odpalovacího lanka - v dvou různých časových okamžicích a složitější konstrukce složený taky jen z lanek a uzlů - všechny tyto ukázky jsou součástí animací, kde 4. rozměr - čas - přidává zase něco navíc.

Pozn.: Obraz je zde inverzně, kvůli původně černému pozadí.

firewire1.png

firewire2.png

flower.png

q3.png

firecone.png

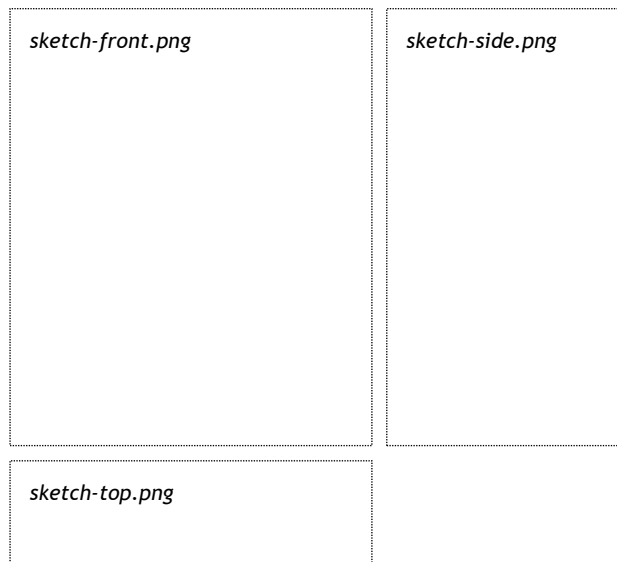
Obr. 7: Ukázky prostředí po vizualizaci

Ukázkový příklad

Jako ukázkový příklad popíšu detailně vytvoření jedné průměrně složité animace.

Nápad

Všechno začíná nápadem a protože teď vytváříme 3D modely tak je vhodnější obrázek než tisíc slov (obr. 8). Jedná se o ohňostroj sestavený z několika málo raket rozložených na mostíku (část kružnice) ze kterého se pak jednotlivé rakety odpalují různým směrem. Aby to bylo zajímavější, směr raket je „modulován“ na tvar sinusoidy.



Obr. 8: Skica nápadu

Implementace

Vhodný postup pro implementaci je sestavení odpalovacího mechanismu a následně pak přidání raket. V našem příkladu jde teda o výpočet jednotlivých vrcholů - jak hlavního odpalovacího lanka (položeném na oblém mostíku) tak výchozích pozic raket. Tento krok zajišťuje následný kód:

```
$a = 1/3;           // angle
$t = 3;           // time of fire
$v = 15;          // half of the width
$n = 16;          // number of elements on the path
$dr = 1;          // radius difference
$eng = array( 400, 200 );
$clr = array( array(1.0,0.0,0.0) // red
              , array(0.0,1.0,0.0) // green
              , array(1.0,0.6,0.0) // orange
              , array(1.0,1.0,0.0) // yellow
              , array(0.8,0.0,0.8) // magenta
            );

$r = $v/sin(1/2*M_PI*$a); // radius
$o = sqrt($r*$r-$v*$v);   // offset
$l = $a*M_PI*$r;         // quarter of circle
$spd = $l/$t;            // fire speed
$pt = $de = array();

for ($i=0;$i<$n;$i++) {
    $re = $i/($n-1); // 0..1
    $ca = M_PI * ( 1/2 - ($a*$re - 1/2*$a) );
    $x = $r*cos($ca);
    $y = $r*sin($ca) - $o;
    $pt[$i] = new BaseVector( 0, $x , $y );
    $x = ($r+$dr)*cos($ca);
    $y = ($r+$dr)*sin($ca) - $o;
    $de[$i] = new BaseVector( 0, $x , $y );
}
```

Když máme body, tak je hračkou sestrotit konstrukci - jde jen o prosté generování objektů a nastavování jejich parametrů. Poslední krok v implementaci je pak určit počátek - co se má vlastně zapálit (v našem případě to je první uzel):

```

$nde = $dly = $ict = $fwr = array();
for ($i=0;$i<$n;$i++) {
    $re = $i/($n-1); // 0..1

    // node
    $universe->addObject( $nde[$i] = new FireNode( $pt[$i], 0 ) );

    // delay
    $universe->addObject( $dly[$i] = new FireWire( $pt[$i], $de[$i], $spd ) );
    $nde[$i]->next[] = $dly[$i];

    // interconnect
    if ($i<($n-1)) {
        $universe->addObject( $ict[$i] = new FireWire( $pt[$i], $pt[$i+1], $spd ) );
        $nde[$i]->next[] = $ict[$i];
    }
    if ($i>0) $ict[$i-1]->next = $nde[$i];
    if (!isset($noFirework)) {

        // ... and some firework
        $fwr[$i] = new FireWork( $de[$i] // position
                                , $de[$i]->sub( $pt[$i] ) // direction
                                ->add( new BaseVector(
                                    sin($re*2*M_PI),0,0 )
                                , 0.2 // mass
                                , 90 // fuelPercent
                                , 0.1 // fDPS
                                , 50 // fSPD
                                );

        // setup
        $fwr[$i]->childResistance = 1/3;
        $fwr[$i]->childLives = 6; // [second]
        $fwr[$i]->energyPerKg = $eng[$i%count($eng)];

        $c = $clr[$i%count($clr)];
        $fwr[$i]->color->r = $c[0];
        $fwr[$i]->color->g = $c[1];
        $fwr[$i]->color->b = $c[2];

        // exists
        $universe->addObject( $fwr[$i] );

        // link da fire! :)
        $dly[$i]->next = $fwr[$i];
    }
}
$nde[0]->activate();

```

Odpalovací mechanismus

Pokud máme model implementován, můžeme zkusit vytvářet animace jako je tato:

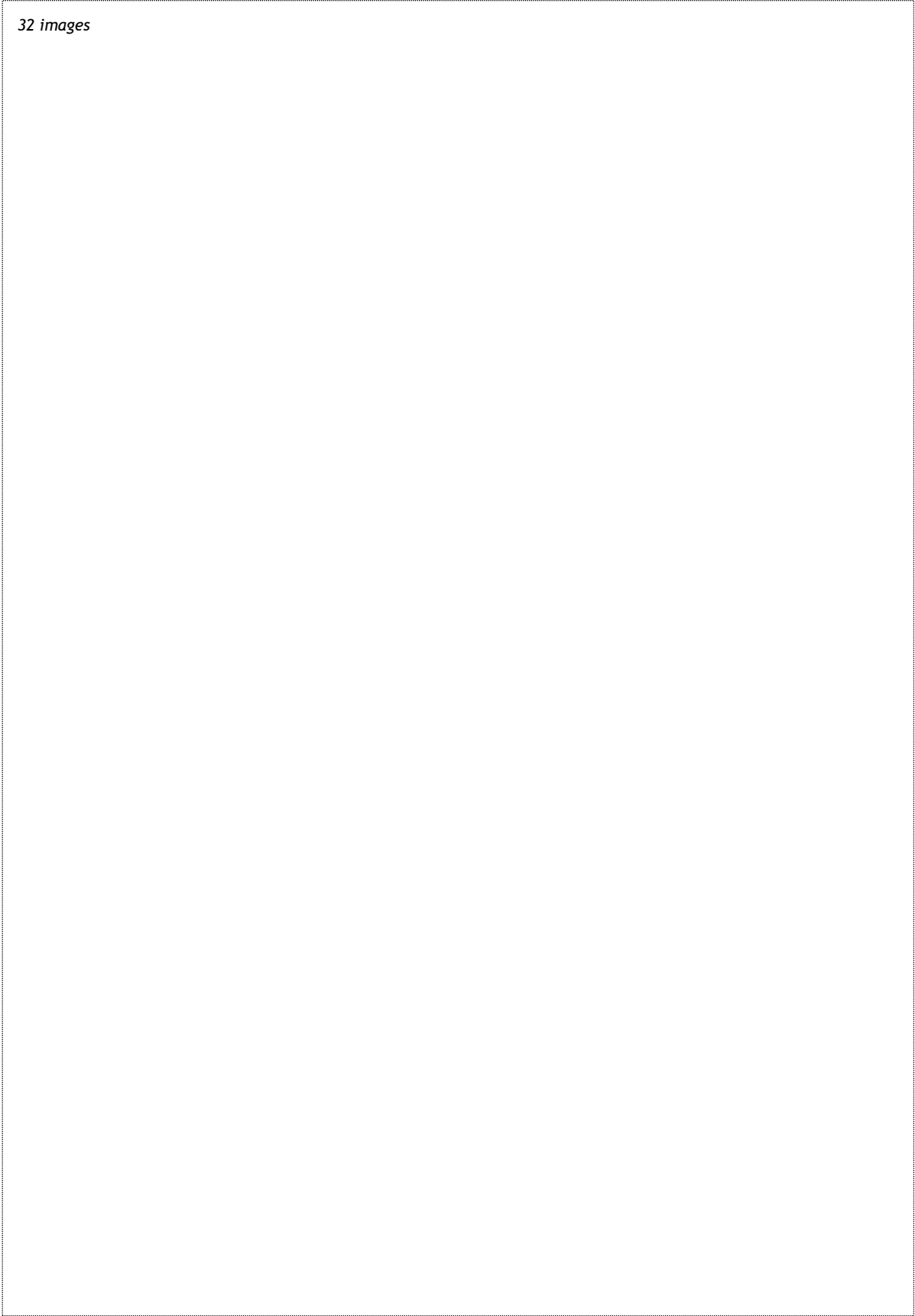


Obr. 9: Animace odpalovacího mechanismu

Výsledná animace

Pro lepší názornost jsem zvětšil počet ohňostrojů (na 32) a zmenšil jejich koeficient výbušnosti. Takto jsem dostal následující animaci:

32 images



Závěr

Projekt byl realizován v časovém rozpětí asi dvou týdnů. Nejprve jsem implementoval třírozměrné prostředí a jeho transformaci na obyčejné snímky a až pak přišli na řadu simulační prvky. Realizace byla docela zajímavá a získal jsem i nové znalosti při tvorbě několika animací. Jak se chýlil projekt ke konci a animace byly složitější (finální animace má 3200 pohybujících se objektů) tak to nějak přestávalo být až tak zajímavé - nejspíš z důvodu dlouhé doby čekání na výsledek.

V budoucnu bych chtěl rozšiřovat svoje prostředí protože si myslím, že 3D vizualizace bude vždy o něco zajímavější než ta obyčejná a chtěl bych teda mít modul který jednoduše zobrazí velké množství dat, nebo spíš data se složitou strukturou. Vstupem tohoto modulu by měl být popis scény nejlépe ve formě XML.

Pokud pak budu mít potřebu něco simulovat, tak bych možná rozšiřoval i simulační část, ale teď už nemám žádný důvod, takže alespoň uvolním zdrojové kódy pro veřejnost a možná se časem objeví nějaký další projekt založený právě na jádru, které jsem vytvořil já.

Použitá literatura

- [1] **Informace o projektu do MSI**
<https://www.fit.vutbr.cz/study/courses/MSI/private/projekt.html>

- [2] **MPlayer - The Movie Player (Documentation)**
7.7. Encoding from multiple input image files (JPEG, PNG, TGA, SGI)
<http://www.mplayerhq.hu/DOCS/HTML/en/menc-feat-enc-images.html>

- [3] **Odpor vzduchu**
<http://pandora.idnes.cz/part/2000/6/18909>

- [4] **3D Camera Transformation**
<http://www.siggraph.org/education/materials/HyperGraph/viewing/view3d/3dview1.htm>

- [5] **Viewing**
<http://www.ncst.ernet.in/apgdst/cgfac/lectures/cgViewing.pdf>