CHALMERS

EvilEngine

En spelmotor till iPhone

Institutionen för Data- och informationsteknik

Viktor Holmqvist
Robin Rye
Sonny Karlsson
Per Åstrand
Marcus Johansson

Chalmers Tekniska Högskola
Göteborg, Sverige 2011
Kandidatarbete/rapport nr
2011:31

18 februari 2012

Sammanfattning

Den här rapporten beskriver utvecklingen av spelmotorn EvilEngine. EvilEngine består av en fysikmotor, en grafikmotor och mindre delar som hanterar interaktion och ljud. Fysikmotorn klarar av newtonsk fysik i 2 dimensioner, kollisioner mellan konvexa polygoner och fluiddynamik. Grafikmotorn renderar polygoner i 2 dimensioner med färger, texturer, ljuseffekter och skuggor. EvilEngine är huvudsakligen utvecklad för Apples plattform iOS men för att göra den så plattformsoberoende som möjligt är stora delar av spelmotorn skrivna i programspråket C. Rapporten beskriver vägen från tidigt koncept till slutgiltig produkt och diskuterar de val och problem som utveckling av en spelmotor innebär.

Nyckelord:
iOS, spelmotor, fysikmotor, grafikmotor, OpenGL ES, smartphone

Innehåll

1 Introduktion **2**
- 1.1 Problembeskrivning . . . 2
- 1.2 Syfte . . . 3
- 1.3 Avgränsningar . . . 3
- 1.4 Att läsa rapporten . . . 3

2 Bakgrund **4**
- 2.1 Utveckling på mobila plattformar . . . 4
  - 2.1.1 Begränsad beräkningskapacitet . . . 4
  - 2.1.2 Fysiskt gränssnitt . . . 4
  - 2.1.3 Användarmönster . . . 5
  - 2.1.4 iOS-enheter liknar konsoler . . . 5
- 2.2 Hårdvaran i Apples iOS-enheter . . . 5
  - 2.2.1 ARMv6-enheter . . . 6
  - 2.2.2 ARMv7-enheter . . . 7
  - 2.2.3 Retina Display . . . 8
  - 2.2.4 Evaluering av kontroller . . . 8
- 2.3 Inspiration . . . 10
  - 2.3.1 Limbo . . . 11
  - 2.3.2 Trine . . . 11
  - 2.3.3 Shift & And Yet It Moves . . . 12
  - 2.3.4 Box2D . . . 13

3 Planering **14**
- 3.1 Utvecklingsmetodik . . . 14
  - 3.1.1 Versionshantering . . . 14
  - 3.1.2 Utvecklingsmiljö . . . 15
- 3.2 Prototyper . . . 15
- 3.3 Granskning . . . 15
- 3.4 Testning . . . 16

4 Teori **17**
4.1 Basbibliotek . . . 17
4.1.1 Datatyper . . . 17
4.1.2 Vektor- och matrisoperationer . . . 18
4.2 Partikelfysik . . . 19
4.2.1 Integration av en partikel per skärmuppdatering . . . 20
4.2.2 Kraftgeneratorer . . . 20
4.3 Stelkroppsfysik . . . 21
4.3.1 Rotation . . . 21
4.4 Fysiken bakom kollisioner . . . 22
4.4.1 Impulser . . . 22
4.4.2 Friktion . . . 23
4.4.3 Separating Axis Theorem . . . 23
4.5 Fluiddynamik . . . 24
4.5.1 Navier-Stokes Ekvationer . . . 24
4.5.2 Fluid i en låda . . . 25
4.5.3 Algoritm . . . 25
4.5.4 Densitetsberäknare . . . 26
4.5.5 Hastighetsberäknare . . . 29
4.5.6 Rendering . . . 31
4.5.7 Utökning och optimering av algoritmerna . . . 32

5 Utförande **34**
5.1 Litteraturstudie . . . 34
5.2 Definition av spelmotorn . . . 35
5.2.1 Struktur och funktionalitet . . . 35
5.2.2 Portabilitet . . . 35
5.2.3 Gränssnitt . . . 36
5.3 Fysik . . . 36
5.3.1 Beteenden . . . 36
5.3.2 Stela kroppar . . . 36
5.3.3 Enkla geometriska figurer . . . 37
5.3.4 Polygoner . . . 38
5.3.5 Optimering . . . 39
5.3.6 SAT . . . 39
5.3.7 Kollisionsberäkning . . . 40
5.3.8 Penetrering . . . 40
5.3.9 Vilande objekt . . . 41
5.3.10 Spekulativa kontakter . . . 41
5.3.11 Matriser . . . 41
5.3.12 Hantering av krafter . . . 42
5.4 Grafik . . . 42
5.4.1 Exempel från Apple . . . 42
5.4.2 Shapes - test av fysik och kollisioner . . . 42

5.4.3 Triangulisering av polygoner . . . 43
5.4.4 Texturer . . . 43
5.4.5 Shaders . . . 44
5.4.6 Omskrivning till C . . . 45
5.4.7 Optimeringar av grafikdata . . . 45
5.4.8 Ljuseffekter . . . 46
5.5 Interaktion . . . 47
5.5.1 Val av kontroller . . . 47
5.5.2 Styra gravitationen med accelerometern . . . 47
5.5.3 Interagera med objekt genom multi-touch . . . 48
5.6 Testning . . . 49
5.6.1 Enhetstester . . . 50
5.6.2 Visuell testning . . . 50
5.7 Optimering . . . 50
5.7.1 Accelerate . . . 51
5.7.2 Optimeringar med assemblyspråk . . . 51

6 Resultat **52**
6.1 Strukturen i EvilEngine . . . 52
6.1.1 Moduler . . . 52
6.1.2 Datatyper . . . 53
6.1.3 API . . . 56
6.2 Vektor- och matrisbibliotek . . . 57
6.3 Prototyper . . . 58
6.3.1 Prototyp 1 . . . 58
6.3.2 Prototyp 2 . . . 58

7 Diskussion **60**
7.1 Planering och arbetsprocess . . . 60
7.2 Val av teknik . . . 61
7.2.1 Objective-C och C . . . 61
7.2.2 OpenGL . . . 61
7.2.3 Externa bibliotek . . . 61
7.3 Ljudmotor . . . 62

8 Framtida arbete **63**
8.1 Spelidén . . . 63
8.2 Ljudmotor . . . 64
8.3 Distribution . . . 65
8.4 Scriptning av banor . . . 65
8.5 Kontinuerlig kollisionsdetektion . . . 65
8.6 Ytterligare optimering . . . 66
8.7 Vatten och rök . . . 66

9 Slutsats **68**

A Matrismultiplikation i assemblyspråk **69**

B Renderare för OpenGL ES 2.0 **72**

Litteraturförteckning **78**

Kapitel 1

Introduktion

Sedan Apple släppte iPhone år 2007 och året därpå lanserade den mobila försäljningstjänsten App Store så har marknaden för mobila applikationer exploderat. Det har gett upphov till en revolution där data och beräkningskraft i allt högre utsträckning blivit bärbar och tillgänglig. Allt från amatörutvecklare till stora företag anslöt sig till revolutionen för att bidra till de över 300 000 applikationer som idag finns att ladda ner från App Store. Med den nya marknaden som har bildats för mobila applikationer har en övergång påbörjats där program, spel och verktyg, som vanligtvis används på datorer, börjat finna sina användningsområden på mobila plattformar. Ett sådant exempel är spelmotorn Unreal Engine [1] som har portats till Apples plattform iOS för att förenkla utvecklingen av spel med sofistikerad 3D-grafik.

Den här rapporten kommer att introducera en ny spelmotor till iOS, EvilEngine, utvecklad av författarna till rapporten. Spelmotorn fokuserar främst på grafiska och fysikaliska beräkningar, men är samtidigt modulärt byggd för att underlätta vidareutveckling efter arbetets slut.

Arbetets fortgång illustreras med demonstrationsbanor, prototyper, som har använts för att undersöka och testa hur väl funktionalitet har implementerats, samt om förändringar eller tillägg har varit nödvändiga.

1.1 Problembeskrivning

Den nya marknaden för mobila applikationer som har växt fram under de senaste åren har lett till stora möjligheter för utvecklare. Emellertid har även intressanta problem och utmaningar uppkommit som behöver lösas för att dessa möjligheter ska kunna utnyttjas fullt ut. Vid utveckling av spel till iOS finns det främst två utmaningar; den begränsade beräkningskraften och den lilla skärmen. Eftersom prestanda i datorer ökat enormt de senaste decennierna har spelbranschen börjat producera allt mer sofistikerade spel. När mobiler i allt högre utsträckning används som spelplattform har fokus

inom spelbranschen sakta börjat vända från prestanda och komplexitet till enkelhet och kreativitet. Det har lett till en uppsjö av spel på App Store och andra mobila försäljningstjänster som ofta är mycket enkla och beroendeframkallande.

Tanken med utvecklingen av EvilEngine har varit att ge utvecklare möjligheten att på ett enkelt sätt skapa spel som är roliga och kreativa, men med fler grafiska och fysikaliska effekter än vad många spel till iOS idag har.

1.2 Syfte

Projektets syfte har varit att utveckla en spelmotor till iOS. Fokus har främst legat på tekniken bakom spelmotorn och dess komponenter; grafik, fysik och interaktion.

Spelmotorn omfattar en grafikmotor med stöd för rendering av 2D-grafik, skuggor, ljuseffekter och partiklar. De fysikaliska beräkningarna för dessa objekts rörelser sker i fysikmotorn, vilken främst inkluderar klassisk mekanik. Förutom en grafik- och fysikmotor har det även implementerats en modul som hanterar interaktion från pekskärm och accelerometer.

1.3 Avgränsningar

Projektet har inte berört utveckling och design av ett komplett spel. Fokus har helt legat på spelmotorn och prototyper som visar upp dess funktionalitet. Stöd för rendering av grafik i 3 dimensioner har inte implemeterats och i fysikmotorn finns det inget stöd för fysiska effekter utöver klassisk mekanik och fluiddynamik. Möjligheterna med att använda gyrometer och kompass som kontroller i ett spel har inte utforskats och stöd för dessa sensorer finns inte i spelmotorn. Spelmotorn har inte stöd för flerspelar-lägen med nätverkskommunikation, inte heller för övriga tekniker som inte ryms inom grafikrendering, fysikberäkningar, interaktion och ljud.

1.4 Att läsa rapporten

Rapporten kommer i tur och ordning beskriva processen från planering till spelmotorns tillstånd vid projektets slut. För den intresserade läsaren har all teori abstraherats och samlats ihop i kapitel 4.

En ordlista som beskriver viktiga begrepp, ord och förkortningar finns i slutet av rapporten.

Kapitel 2

Bakgrund

I detta kapitel kommer utmaningarna med utveckling till en mobil plattform beskrivas, samt hur iOS liknar och skiljer sig från andra vanliga plattformar för spelutveckling. De uppsättningar av hårdvara som finns i iOS-enheter kommer tas upp och det kommer undersökas hur de kan utnyttjas effektivt. Det kommer dessutom nämnas några spel som inspiration har hämtats ifrån.

2.1 Utveckling på mobila plattformar

Fokus för detta projekt är, som nämnts tidigare, utvecklingen av en spelmotor på iOS-baserade enheter. iOS utgör en av två stora mobila plattformar som används idag, Android utgör den andra. För att ge perspektiv på utmaningar för utveckling till mobila plattformar beskrivs under denna sektion först gemensamma utmaningar för mobil utveckling och sedan likheter mellan utveckling för iOS och för spelkonsoler.

2.1.1 Begränsad beräkningskapacitet

För spelutveckling är det ofta viktigare att spelet är roligt och engagerande än att det är grafiskt avancerat, men då valet att utveckla en spelmotor har gjorts finns ett mycket större fokus på teknik, som fysikberäkningar och grafikrendering. Dessa områden bygger generellt på tunga beräkningar och därmed begränsas spelmotorer ofta av plattformens beräkningskapacitet. Eftersom mobila enheter ofta designas för låg strömförbrukning framför prestanda begränsas vi ytterligare.

2.1.2 Fysiskt gränssnitt

En stor utmaning specifikt för alla iOS- och många andra moderna mobila enheter är avsaknaden av knappar. Istället används skärmen som fysiskt gränssitt. Detta ger unika utmaningar för spelutveckling. Det kräver i sin tur att större hänsyn behöver tas till vad som ska finnas på skärmen. Avsaknaden

utav standardiserad användning av skärmen som fysiskt gränssnitt kräver visuella ledtrådar så som knappar och kan leda till att spelupplevelsen blir lidande.

Många av dessa mobila enheter kommer dock även med andra fysiska gränssnitt i form av olika sensorer såsom gyro och accelerometer för att avläsa rörelser och orientation. Om dessa sensorer används kreativt kan annorlunda upplevelser än de vanligen associerade med spel uppnås.

2.1.3 Användarmönster

Spelande på konsoler och persondatorer kan för varje pass vara i timmar, vilket gör att spelupplevelsen inte lider av begränsade möjligheter till att spara framsteg. I vissa spel kan detta till och med bli delmål. För spel på mobila enheter ser användarmönstret mycket annorlunda ut. Varje speltillfälle är tidsmässigt generellt kortare, men sker oftare, t.ex. som tidsfördriv på bussen. I en undersökning utförd 2009 av Juul svarade 81% av de tillfrågade att det var mycket viktigt att kunna avbryta ett spel [19]. Det är vanligtvis viktigt vid speldesign, således bör en spelmotor möjliggöra avbrott och återupptagande av spel snabbt och effektivt.

2.1.4 iOS-enheter liknar konsoler

Konsoler baseras uteslutande på en uppsättning mjukvara parat med en uppsättning hårdvara. Förutom att konsoler designas för spel så gör denna brist på variation från enhet till enhet att hårdvaran används effektivare av spel. Optimering för en specifik uppsättning hårdvara påverkar alla användare och är därmed ett relativt billigt sätt att göra spel bättre. Spel för konsoler ser därför ofta bättre ut än spel på jämförbara persondatorer tack vare att spelen använder hårdvaran effektivare.

iOS-baserade enheter har en uppsättning mjukvara och en handfull uppsättningar hårdvara. Detta gör att samma fördel vid optimering erhålls för dessa enheter. Därför har några av dessa möjligheter undersökts i projektet.

2.2 Hårdvaran i Apples iOS-enheter

Det finns idag ett flertal iOS-enheter ute på marknaden. Gemensamt för alla enheter är att de har en ARM-processor (Advanced RISC Machine) parat med en grafik-processor från PowerVR.

ARM är en RISC-design (Reduced Instruction Set Computing) vilket innebär att det finns ett fåtal isolerade instruktioner för att ladda in till och spara data från register medan alla andra instruktioner utförs register till register. Detta gör att RISC kräver mindre logik när de implementeras i hårdvara och instruktioner kan i regel utföras i en klockcykel. Detta kan jämföras med CISC-arkitekturer där en instruktion ofta kan ladda och spara

data till minne samtidigt som någon operation utförs. Detta gör att färre instruktioner krävs per program, men i gengäld är ofta processorerna mer komplexa och varje instruktion kräver i regel fler än en klockcykel. Anledningen till att RISC ofta återfinns i inbäddade och mobila tillämpningar är att designen ger bättre prestanda per watt än motsvarande CISC. Det finns två generationer iOS-enheter idag, baserade på olika versioner utav ARM-arkitekturen.

Den första generationen introducerades med iPhone 2G, Apples första mobiltelefon, år 2007, och baseras på en ARMv6-processor med en PowerVR MBX grafik-processor. Den andra generationen introducerades med iPhone 3GS, år 2009, och baseras på en ARMv7-processor tillsammans med en PowerVR SGX grafik-processor.

Model	iPhone 2G	iPhone 3G	iPhone 3GS	iPhone 4
Processor	ARM11	ARM11	Cortex-A8	Cortex-A8
Arkitektur	ARMv6	ARMv6	ARMv7	ARMv7
PowerVR krets	MBX lite	MBX lite	SGX	SGX

Tabell 2.1: Tabell över iPhone modeller.

Det bör även tilläggas att den nyligen introducerade iPad 2 med Apples A5 använder sig av en flerkärnig ARMv7-processor tillsammans med en flerkärnig PowerVR SGX kallad PowerVR SGXMP. Detta kan vara av intresse vid framtida arbete på spelmotorn då det ger en signifikant prestandaökning som skulle kunna vara till fördel för oss.

2.2.1 ARMv6-enheter

Enheter baserade på ARMv6 är förutom iPhone 2G och iPhone 3G även första och andra generationen iPod Touch. Processorn som används i dessa enheter heter ARM11 och full dokumentation om denna finns på [2].

Flyttal

I de ARMv6-processorer som används utav Apple finns alltid en VFP-enhet. Detta gäller inte för alla ARMv6-processorer då detta är en valbar enhet. VFP utför flyttalsoperationer på tal med både enkel- och dubbelprecision i hårdvara och är implementerat som en hjälpprocessor med egna flyttalsregister. Trots att VFP står för Vector Floating Point och enheten stödjer operationer på korta vektorer så utförs dessa sekventiellt på flyttalen istället för parallellt.

Thumb-instruktioner

För att öka koddensiteten kan processorn använda sig utav ett instruktionsset vid namn Thumb. Dessa instruktioner tar endast upp 16 bitar istället

för de 32 bitar som används av ARM-instruktioner vilket gör att färre laddningsoperationer behövs för att utföra program med Thumb-instruktioner än ARM-instruktioner.

Nackdelarna med Thumb är att färre antal register kan användas och att endast ett fåtal instruktioner kan utföras villkorligt i jämförelse med ARM där nästan alla instruktioner kan vara villkorliga, samt att flyttalsinstruktioner inte är tillgängliga. För ett program som använder sig utav Thumb krävs det därför i regel fler instruktioner än motsvarande program utan, men den ökade densiteten gör ändå att Thumb körs snabbare. Detta gäller dock inte för flyttalsintensiva program då dessa behöver byta mellan Thumb- och ARM-instruktioner varje gång flyttalsoperationer ska utföras, vilket gör att dessa i regel är långsammare än motsvarande program med endast ARM-instruktioner trots ökad densitet.

OpenGL ES 1.1

Med alla ARMv6-baserade enheter inkluderas en PowerVR MBX grafikprocessor. Denna grafik-processor stödjer i iOS OpenGL ES 1.1, hädanefter förkortat som ES1. ES1 är baserad på OpenGL 1.5 för stationära datorer och använder sig utav en FFP (Fixed Function Pipeline). Detta innebär att ES1 definierar ett begränsat antal funktioner för att manipulera hur grafik ritas på skärmen och att dem är de enda sätten som ritningen på skärmen kan påverkas av. Detta har som konsekvens att om en teknik inte finns med i ES1 så finns det inget sätt att simulera den i grafikprocessorn. Dokumentation på ES1 finns på [3].

2.2.2 ARMv7-enheter

Enheter baserade på ARMv7 är förutom iPhone 3GS och iPhone 4 även iPad samt tredje och fjärde generationens iPod Touch. För oss relevanta förändringar gentemot ARMv6 rör framförallt flyttalsberäkningar. Stödet för Advanced SIMD, marknadsfört som NEON, förbättrar markant prestandan vid flyttalsberäkningar med enkelprecision. Thumb-2, en utökning utav Thumb från ARMv6, ger åtkomst till flyttalsinstruktioner utan att behöva byta till ARM-instruktioner. Processorn som används heter Cortex-A8 och full dokumentation finns på [4].

Flyttal

I de ARMv7-baserade enheterna från Apple ingår stöd för NEON, vilket är ett set SIMD-instruktioner för decimaltal av 16, 32 eller 64 bitar och flyttal med enkelprecision. SIMD står för Single Instruction Multiple Data och innebär att att en instruktion kan operera på flera data. NEON-instruktionerna utför operationer på antingen 64 eller 128 bitar data åt gången, vilket är två eller fyra flyttal med enkelprecision. NEON-instruktioner utförs på en

hjälpprocessor och vissa kombinationer av instruktioner kan utföras parvis om inga databeroenden finns. I Apples enheter finns utöver NEON även stöd för VFP och det är denna som används för flyttalsberäkningar efter att kompilatorn har översatt kod i ett högnivåspråk till maskinkod.

Thumb-2-instruktioner

Thumb-2 bygger på Thumb från ARMv6 och är bakåt-kompatibel. Den utökar Thumb med ett antal 32-bit-instruktioner för bland annat flyttalsoperationer. Detta medför att även flyttalsintensiva program kan dra nytta av den ökade koddensiteten utan att behöva byta till ARM-instruktioner.

OpenGL ES 2.0

I ARMv7-baserade enheter används en PowerVR SGX grafik-processor som har stöd för både OpenGL ES 1.1 och OpenGL ES 2.0 i iOS. OpenGL ES 2.0, hädanefter förkortat som ES2, skiljer sig markant från ES1 och är inte bakåt-kompatibelt. ES2 baseras på OpenGL 2.0 för stationära datorer och bygger på en programmerbar pipeline. Detta innebär att programmeraren kan påverka vissa steg i renderingen genom att ladda upp shader-program som bestämmer hur modellerna ska manipuleras när de ritas ut på skärmen. Detta ökar flexibiliteten i renderingen och nya tekniker kan användas även om dessa tekniker är nyare än ES2. Dokumentation på ES2 finns på [5].

2.2.3 Retina Display

Med iPhone 4 introducerade Apple en ny skärm med samma dimensioner som tidigare men med dubbelt så hög upplösning. Denna skärm kallas för Retina Display. När program ej gjorda för denna upplösning används på en iPhone 4 skalar iOS grafiken för att täcka alla punkter på skärmen. För att utnyttja denna upplösning utan skalning krävs det att kod för att detektera storleken på ritytan används för att ladda texturer med hög upplösning samt att tala om för systemet att inte skala om grafiken.

2.2.4 Evaluering av kontroller

Multi-touch

Alla Apples varianter av iPhone, iPad och iPod Touch har en s.k. multi-touch-skärm. Multi-touch används i mycket stor utsträckning, bl.a. till menynavigation, virtuellt tangentbord och att zooma in och ut på skärmen. De fysiska knapparna som sitter på enheterna är hemskärms-, on/off- och ljudkontroll-knapparna. I fig. 2.1 nedan syns principen för en multi-touch-skärm. För varje finger användaren trycker på skärmen genereras en koordinat som kan användas för att styra program och spel. Det som begränsar

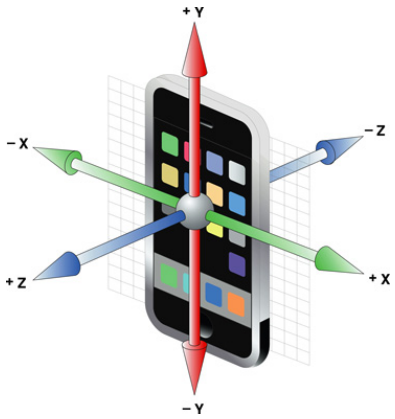
antalet fingrar som kan detekteras samtidigt är antalet fingrar som ryms på skärmen.

Figur 2.1: Multi-touch skärm med tre fingertryckningar

Genom en panel som är laminerad på glaset känner enheten av beröringar med hjälp av elektriska fält. Panelen överför sedan informationen till den underliggande skärmen. Eftersom den använder sig av elektriska fält kan inte vad som helst användas till att simulera tryckningar.

Accelerometer

Precis som med multi-touch finns en s.k. accelerometer inbyggd i samtliga iOS enheter. Accelerometer används till att mäta accelerationer i X, Y och Z-led, se fig. 2.2. Genom att man lutar eller förflyttar enheten uppstår en acceleration som kan avläsas som en vektor. I fallet då man lutar enheten så är det gravitationen som skapar en acceleration.

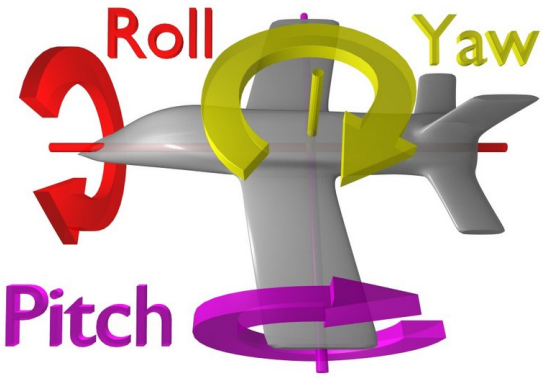


Figur 2.2: Axlarna som accelerometern mäter av

Accelerometern används flitigt i många spel för att t.ex. simulera en bilratt eller gravitation.

Gyroskop

Gyroskopet introducerades med iPhone 4 och finns numera även i iPad 2 och 4:e generationens iPod touch. Ett gyroskop används för att mäta rotationer runt X-, Y- och Z-axlarna, även kallat "roll", "yaw" och "pitch", se fig. 2.3. Tillsammans med accelerometern bildar gyroskopet en komplett 6-axlad rörelse-avläsning som väldigt exakt kan bestämma hur enheten är vinklad, och i vilken riktning.



Figur 2.3: Rotationer runt axlarna visualiserat med hjälp av ett flygplan

Gyroskopet används idag inte i så stor omfattning i applikationer, men förväntas bli en stor del av spelmarknaden framöver.

Kompass och GPS

Andra typer av kontroller som skulle kunna avläsas är kompassen och GPS-sensorn. En kompass finns inbyggd i enheter som kom efter lanseringen av iPhone 3GS. Kompassen skulle bland annat kunna användas tillsammans med accelerometern för att simulera ett gyroskop. Det skulle givetvis inte ge alls samma precision som ett riktigt gyroskop, men kompassen finns tillgänglig i ett betydande antal fler enheter än gyroskopet. Man skulle även kunna utveckla ett spel som använder sig av GPS till förflyttning i spelet.

2.3 Inspiration

För att bättre förstå vad som kan behövas i en spelmotor har ett flertal spel granskats lite närmare och det intressanta i dem plockats ut. Gemensamt

för dem alla är att man styr en karaktär i världen och att spellogiken är i 2D.

2.3.1 Limbo

Limbo [13] är det första spelet av den danska spelutvecklaren Playdead.

Miljön och färgsättningen i Limbo är relativt enkel men ser ändå välgjord ut, vilket kan vara ganska sällsynt för spel som endast har färgsatts med gråskala. Med få dialoger, effektfull tystnad och denna mörka färgsättningen kan spelet framstå som dystert. Figuren 2.4 är hämtad ur en spelsekvens i spelet.

De naturliga rörelserna hos karaktärerna och en spelvärld med silhuetter uppdelade i olika lager har inspirerat oss att fokusera vår spelmotor på enklare grafik men avancerad fysik.

Figur 2.4: Limbo

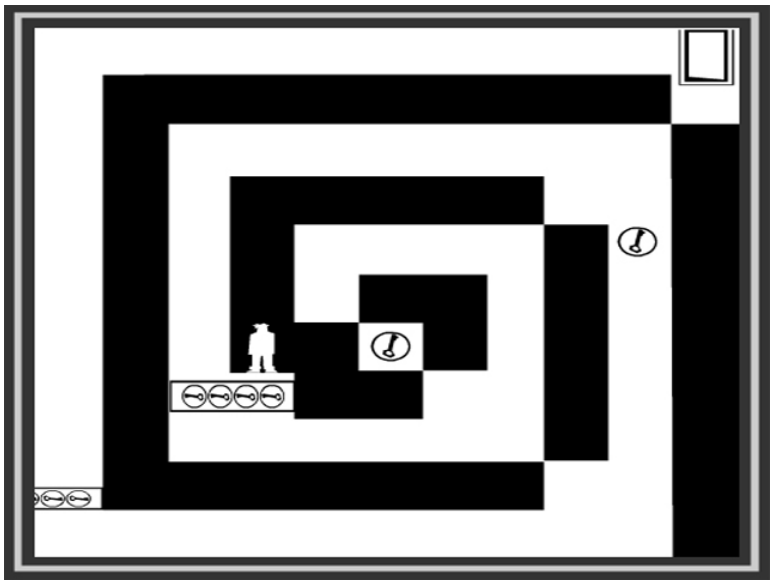
2.3.2 Trine

I Trine kan spelaren skifta mellan tre olika karaktärer med olika förmågor för att lösa pussel och ta sig förbi hinder. Det intressanta är att spelets logik är i 2D men renderas med perspektiv i 3D. Detta var något som vi var intresserade av att inkludera i vår spelmotor om tid hade funnits. Nedanför i figur 2.5 syns en bild ur spelet.

Figur 2.5: Trine

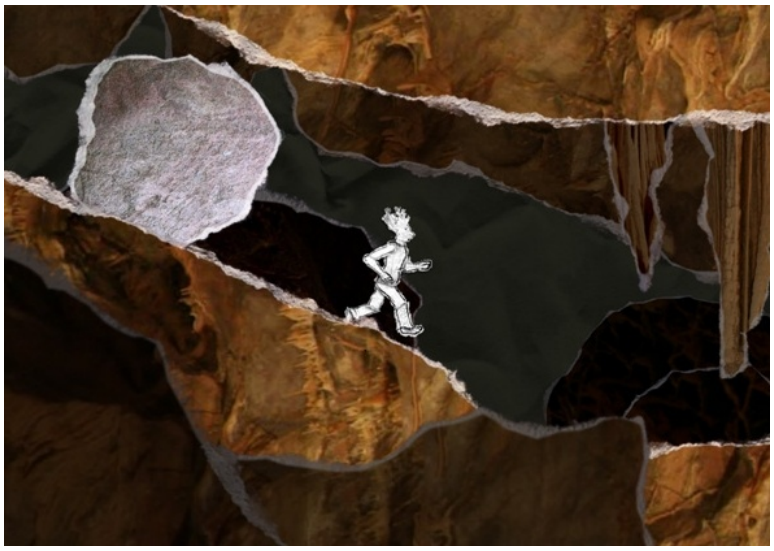
2.3.3 Shift & And Yet It Moves

Spelet Shift, i figur 2.6 nedan, går ut på att ta sig ut från olika rum genom att vända spelvärlden. Även And Yet It Moves (hädanefter AYIM), figur 2.7, går ut på att ta sig från punkt A till B.



Figur 2.6: Shift

En central idé i vårt projekt har varit att använda accelerometern för att rotera spelvärlden, och därför har vi tittat på hur fast rotation fungerar vid problemlösning. Shift har fast rotation på 180 grader vid varje rotation och AYIM har fast rotation på 90 grader. I båda spelen är gravitationsskiften det centrala verktyget för att lösa de pussel man ställs inför.



Figur 2.7: And Yet It Moves

2.3.4 Box2D

Box2D [17] är en fysikmotor ursprungligen skriven utav Erin Catto, men släpptes under en öppen källkods-licens år 2007. Ett antal fysikmotorer och spel bygger på denna kod, framförallt det mycket populära Angry Birds.

Framförallt använder sig fysikmotorn av en snabb och effektiv lösning av kollisioner mellan konvexa polygoner, cirklar och kant-figurer. Projektet har inspirerats av hur kollisionshanteringen utförs i 3 faser, en första approximativ kollisionsdetektion, en andra mer noggrann kollisionsdetektion och en tredje fas där egenskaper för kollisionerna räknas ut.

Kapitel 3

Planering

För att sammansvetsa varje gruppmedlems mål och idéer till ett gemensamt mål för gruppen, så gjordes innan utvecklingens början en planering. I planeringen diskuterades det fram hur arbetet skulle fungera, en tidsplan, samt vilka verktyg och metoder som skulle användas. Vad vi kom fram till under planeringsprocessen kommer tas upp i detta kapitel.

3.1 Utvecklingsmetodik

Utvecklingen har skett efter en iterativ modell som huvudsakligen var uppdelad i två stora och flera mindre iterationer. Genom regelbundna gruppmöten har planering kring vad som ska göras, diskussioner kring vad som gjorts, samt utbyte av annan relevant information skett. Mötena har varit användbara för att få en överblick över projektet och förbättra idéer. Förutom dessa gruppmöten har även möten med handledaren inträffat för att diskutera frågor relaterade till projektet.

För att förmedla information inom gruppen utanför mötena har en privat blogg använts där medlemmar delat med sig av kunskap, mötesprotokoll och reflektioner. Vissa av dessa inlägg har dessutom legat till grund för ett antal avsnitt som återfinns i denna rapport.

Dropbox har används för fildelning och en maillista har använts för att snabbt kommunicera mellan gruppmedlemmar.

3.1.1 Versionshantering

För att underlätta utvecklingen har användning av versionshantering tillämpats. Det har medfört att alla i gruppen alltid har haft en uppdaterad version av kodbasen tillgänglig. Det underliggande systemet heter subversion, men ett system för projekthantering ovanpå detta som heter trac har också använts.

Trac har inbyggd funktionalitet för att hantera uppgifter, så kallade tickets. Genom dessa kan en medlem lägga till ett förslag på en funktion, en buggrapport eller något annat som behöver göras i projektet. Sedan har medlemmar kunnat acceptera uppgifter eller tilldela uppgifter till andra gruppmedlemmar. På så vis har det funnits en bra överblick över vad varje person gjort och vad som funnits kvar att göra. Efter att uppgiften slutförts har resultatet delats med gruppen genom subversion och trac tillsammans med en referens till ursprungliga uppgiften och en beskrivning av de ändringar som gjorts.

3.1.2 Utvecklingsmiljö

Implementationen av spelmotorn har skett med C och Objective-C. Som utvecklingsmiljö har Apples Xcode använts, som är det IDE som vanligtvis används vid utveckling till Mac OS X och iOS.

3.2 Prototyper

För att visa vad som implementerats i spelmotorn, bestämdes att demonstrationsbanor skulle utvecklas. Dessa skulle vara relativt enkla banor där spelmotorns funktionalitet kunde testas visuellt. Dessa banor demonstrerades dessutom på redovisningarna, vilket har gjort det enklare för andra personer att förstå vad vårt projekt har omfattat.

Demonstrationsbanorna skulle även fungera som väldigt simpla spelprototyper, för att ge en indikation för vilken funktionalitet som behöver läggas till för att möjliggöra att skapandet av ett underhållande spel ovanpå spelmotorn.

3.3 Granskning

För att förbättra kvalitén i koden och rapporten så har granskning av varandras arbete utförts. Granskningen har gått ut på att läsa varandras texter och kod och ge återkoppling eller förslag på tillägg och ändringar. På detta sätt har problem eller buggar kunnat upptäckas och åtgärdas tidigt. Även ändringar och tillägg har kunnat förmedlas mellan gruppmedlemmarna vilket gett bredare stöd hos gruppen för förändringar och slutsatser.

Inom projektets delområden har dessutom huvud- och bi-ansvariga utsetts, vars uppgift har varit att se till att arbetet gått framåt, dessutom har de haft huvudsakligt ansvar för sina respektive avsnitt i rapporten.

För att få ett annat perspektiv utifrån så läste även handledaren och en person från fackspråk det som skrivits för att komma med synpunkter och ge återkoppling. Även utbyte med andra kandidatgrupper har skett där de

har läst igenom rapporten och kommit med kommentarer, samtidigt som vi har läst deras och gett feedback.

När vi granskat rapporten inom gruppen, så har vi utgått ifrån de betygskriterier och riktlinjer som funnits att tillgå.

3.4 Testning

Under projektets gång genomfördes testning för att se att koden fungerade som den skulle. Den typ av test som genomfördes var "Unit Testing" och det ramverk som användes var OCUnit, vilket är ett ramverk som finns inbyggt i Xcode.

Kapitel 4

Teori

I detta kapitel beskrivs grundläggande datatyper som har använts i spelmotorn samt detaljer för hur implementationen av dessa har gjorts. Här beskrivs också detaljerna för de fysikaliska formlerna och koncepten som har använts när spelfysiken har implementerats.

4.1 Basbibliotek

Eftersom det är önskvärt att spelmotorn ska vara så plattformsoberoende som möjligt så gjordes valet att implementera så mycket som möjligt i programspråket C.

4.1.1 Datatyper

Under implementeringen av bland annat grafikmotorn uppkom ganska snabbt ett behov för enkla datatyper som inte finns i standardbiblioteket i C, t.ex. dynamiska fält, länkade listor och hashtabeller.

Dynamiska lista

Dynamiska listor är bra när man inte vet hur många element en lista kommer innehålla och om man vill kunna lägga till element dynamiskt, som namnet antyder. I standardbiblioteket i C finns statiska fält som initieras med en viss kapacitet. För att utöka ett statiskt fält behöver man allokera mer minne. För att hålla koll på när fältet behöver utökas behöver man också veta nuvarande kapacitet och antalet tagna positioner. Denna funktionalitet har kapslats in i en struktur med funktioner som lägger till och tar bort element och utökar fältet automatiskt när det behövs.

Hashtabeller

Hashtabeller använder en hashfunktion för insättning och uppslagning av ett element, vilket medför att uppslagning är väldigt snabb. Insättning är inte alltid lika snabbt p.g.a. att hashfunktionen inte är perfekt och att en insättning kan innebära kollisioner. Vi använder en variant av *Cuckoo Hashing* [8] i vår hashtabell.

Binär heap

En binär heap är ett partiellt ordnat vänsterbalanserat träd, detta innebär att det är fullt i alla nivåer utom eventuellt den djupaste nivån som i sådana fall är fylld från vänster. Att det är partiellt ordnat innebär att en nod alltid är sorterad mot sina barn och sin förälder. Ordningen varierar beroende på vilken sortering (min eller max vanligtvis) man är ute efter.

För att göra typen flexibel inkluderas en funktionspekare och alla element har typen void-pekare. Det första innebär att när man skapar en heap så skickar man med en adress till en funktion, denna funktionens uppgift är att jämföra två element i trädstrukturen för att bestämma deras prioritet och därmed deras position.

Denna implementation kommer innebära att heapen t.ex. kan användas för en enkel int eller pekare till stora strukturer. Oavsett vad som finns på heapen så skickas det alltid med en funktionspekare som säger hur typen ska sorteras.

Det huvudsakliga användningsområdet för heapen är vid implementering utav snabba prioritetsköer.

4.1.2 Vektor- och matrisoperationer

För att kunna utföra de beräkningar som behövs för att simulera och rita fysik och grafik behövs det tillgång till grundläggande vektor- och matrisoperationer. Det finns färdiga bibliotek som ger tillgång till dessa, men för att lära oss så mycket som möjligt så gjordes valet att implementera egna bibliotek.

Vektorbiblioteket innehåller en definition av vektorer med flyttal i 2 dimensioner. En vektor representeras bara av en slutpunkt men antas utgå från origo. Biblioteket innehåller operationer för att beräkna längd och riktning för vektorer, addition, subtraktion, skalärprodukt och determinant av två vektorer samt multiplikation med skalär och normalisering av en vektor.

Matrisbiblioteket definierar en 4x4 matris som består av en lista med 16 flyttal. Anledningen till denna storlek används är att multiplikation med dessa kan representera all sorts rörelse i 3 dimensioner, vilket är mycket användbart vid grafik- och fysik-sammanhang. Operationer för att multiplicera matriser med varandra och vektorer samt transponera en matris finns i bib-

lioteket. Det finns även operationer för att skapa matriser identitetsmatriser och matriser som ger rotation, skalning eller förflyttning.

Under utvecklingens gång har vektor- och matrisbiblioteken modifierats och optimerats flertalet gånger för att stämma överens med specifikation och prestandrakrav. Prestanda är extra viktigt när det gäller vektor- och matrisoperationerna då de används i stort sett överallt och i många fall vid varje skärmuppdatering - 60 gånger per sekund

4.2 Partikelfysik

När fysikmotorn påbörjades blev det naturligt att först implementera partiklar. Vår definition av en partikel är en punkt med en massa men utan utsträckning. Eftersom en partikel saknar utsträckning behöver man inte tänka på rotationsrörelser vilket förenklar de matematiska formlerna avsevärt.

Partiklar representerades utav strukturer med två flyttal och fyra vektorer i vår kod. Flyttalen för invers massa och dämningsfaktor. Vektorerna för position, hastighet, acceleration samt partikelns ackumulerade krafter [6]. Statiska partiklar blir enkla att representera genom att sätta invers massa till 0, vilket betyder oändligt tungt. Invers massa används också i större utsträckning i våra uträkningar.

Det finns operationer för att integrera partikeln över tid och nollställa ackumulerade krafter. Integrationen av partikeln tas upp mer detaljerat i 4.2.1.

Eftersom endast hänsyn till linjära rörelser behöver göras kan partikelns rörelse beskrivas av Newtons första och andra lag:

1. *Ett objekt fortsätter med en konstant hastighet utan påverkan av externa krafter.*

2. *En kraft som verkar på ett objekt producerar en acceleration som är proportionell mot objektets massa.*

Den nya positionen p' kan därför beräknas med:

$$p' = p + vt + \frac{1}{2}at^2$$

där p, v, a och t är föregående position, hastighet, acceleration respektve uppdateringsintervall. Då integrering av partikeln över ett väldigt litet tidsintervall (0.0167s) utförs kommer bidraget från accelerationen vara försumbart:

$$p' = p + vt \tag{4.1}$$

Den nya hastigheten beräknas på liknande sätt:

$$v' = v + at$$

För att undvika att hastigheten går mot oändligheten på grund av precisionsfel vid användning av flyttal introduceras en dämpningsfaktor d som beror på uppdateringsintervallet:

$$v' = vd^t + at \tag{4.2}$$

Accelerationen härleds från Newtons 2:a lag:

$$f_{ack} = ma$$

där f_{ack} är de ackumulerade krafterna och m partikelns massa. Accelerationen a bryts sedan ut och nu syns det hur inversmassan kommer till användning:

$$a = \frac{1}{m} f_{ack} \tag{4.3}$$

Partiklar med oändlig massa kommer alltså inte att accelereras då inversmassan är 0. När en kraft f appliceras på partikeln adderas den till de accumulerade krafterna:

$$f_{ack} = f_{ack} + f$$

Ekvation (4.1), (4.2) och (4.3) är allt som används för att beskriva en partikels rörelse [9]. Samma ekvationer används även för att beskriva linjära rörelser för stela kroppar.

4.2.1 Integration av en partikel per skärmuppdatering

Tanken med att integrera en partikel per skärmuppdatering är att partikeln alltid ska röra sig på samma sätt, oberoende av uppdaterings-frekvensen. Integratorn tar in uppdaterings-tiden som ett argument och använder sedan denna som tiden i ekvationerna härledda i 4.2 för att uppdatera partikelns position och hastighet. När integrationen är klar anropas operationen för att nollställa ackumulerade krafter. Detta innebär att den på något sätt måste hålla reda på vilka krafter som ska verka på en partikel under en längre period. För att lösa detta används kraftgeneratorer.

4.2.2 Kraftgeneratorer

En kraftgenerator är ett förbestämt sätt att applicera en kraft, de konstanter som behövs för att beräkna kraften och de partiklar/kroppar som kraften ska appliceras på. Kraftgeneratorer används för att veta vilka krafter som ska appliceras på en partikel innan partikeln integreras vid varje skärmuppdatering. Kraftgeneratorer kan användas till många olika sorters krafter, t.ex. gravitation, fjäderkrafter och motstånd som beror på vad för material partikeln passerar genom.

4.3 Stelkroppsfysik

Efter partikelfysiken blev nästa steg stela kroppar. Stela kroppar delar mycket av sitt beteende med partiklar. Den största skillnaden är att stela kroppar har en utsträckning och därmed måste kunna rotera. Kodmässigt innebär detta att ett antal nya variabler behöver läggas till. Vinkeln på ett objekt, förändringen av vinkeln, krafter som verkar i någon annan punkt än i jämviktspunkten och kroppens tröghetsmoment behöver kunna representeras. Alla dessa storheter kan i 2 dimensioner representeras av flyttal. Istället för att lagra de ackumulerade krafterna direkt beräknas istället ett ackumulerat vridande moment som sedan används för att beräkna partikelns nya vinkelhastighet. Av samma anledning som invers massa lagras för partiklar, lagras också inversen av tröghetsmomentet för stela kroppar, alltså att oändlig tröghet enkelt kan representeras.

4.3.1 Rotation

För att beräkna den linjära rörelsen för en stel kropp kan precis samma ekvationer som för en partikel användas. I och med att rotationsrörelser läggs till introduceras ett par nya ekvationer. Den nya vinkeln θ' och vinkelhastigheten ω' kan beräknas med:

$$\theta' = \theta + \omega t \tag{4.4}$$

$$\omega' = \omega + \alpha t$$

där θ, ω och α är föregående vinkel, vinkelhastighet respektive vinkelacceleration. Även här introduceras en dämpning så att vinkelhastigheten inte ska gå mot oändligheten:

$$\omega' = \omega d^t + \alpha t \tag{4.5}$$

Vinkelaccelerationen α kan härledas ur ekvationen för rörelsemängdsmoment L:

$$L = I\omega = I\dot{\theta}$$

Derivering av båda sidor sker för att få det ackumulerade vridande momentet τ_{ack} och eftersom tröghetsmomentet I är konstant kan det brytas ut ur derivatan:

$$\tau_{ack} = \frac{dL}{dt} = \frac{dI\dot{\theta}}{dt} = I\ddot{\theta} = I\alpha$$

Slutligen bryts vinkelaccelerationen ut och även här syns det nu hur nytta kan dras av att lagra inversen av tröghetsmomentet för att underlätta beräkningarna:

$$\alpha = \frac{1}{I}\tau_{ack} \tag{4.6}$$

När en kraft appliceras på en stel kropp måste kraften adderas till de ackumulerade krafterna för att kunna beräkna den linjära påverkan på kroppens

acceleration. Beräkning om en kraft f_p ger upphov till något vridande moment τ när den appliceras i en viss punkt p behöver också utföras.

$$\tau = r_p \diamond f_p \tag{4.7}$$

där $\diamond$ är den vinkelräta skalärprodukten (analogt med kryssprodukt i 3 dimensioner) och r_p är vektorn från kroppens tyngdpunkt till punkten p. Det genererade vridande momentet τ adderas sedan till det ackumulerade:

$$\tau_{ack} = \tau_{ack} + \tau$$

Tillsammans med ekvation (4.1), (4.2) och (4.3) för linjära rörelser i avsnitt 4.2 samt ekvation (4.4), (4.5) och (4.6) kan nu en fullständig beskrivning av en stel kropps rörelser i 2 dimensioner utföras.

4.4 Fysiken bakom kollisioner

I avsnitt 4.3 tas det upp hur en stel kropps rörelser kan beskrivas fullständigt och implementeras. En av de viktigaste aspekterna i en fysikmotor är fortfarande kvar, hur objekt beter sig när de påverkas av andra objekt - kollisioner. Detta avsnitt tar upp hur kollisioner beskrivs och hanteras i fysikmotorn utvecklad i detta projekt.

Kollisioner mellan objekt är en av de större utmaningarna när det gäller att simulera fysik. Utmaningen ligger i att kollisioner sker på en väldigt kort tidsperiod och att väldigt stora krafter är inblandade. I verkligheten pressas två objekt som kolliderar samman och deformeras för att sedan stötas ifrån varandra och återfå sin ursprungliga form beroende på materialens egenskaper [11]. Hela denna process tar bara någon bråkdel av en sekund, involverar ofta väldigt små avstånd och väldigt stora krafter, därför är den alldeles för komplicerad för att simulera på ett bra sätt [6]. För att förenkla beskrivningen av kollisioner används därför ofta en annan storhet istället för kraft – impuls.

4.4.1 Impulser

En kraft påverkar ett objekts acceleration som i sin tur påverkar hastigheten på ett objekt. En impuls är en direkt förändring av hastigheten på ett objekt och behöver därför inte integreras fram över en viss tid, utan kan appliceras direkt vid tillfället för kollisionen. En impuls påverkar både en stel kropps linjära hastighet och dess rotationshastighet. Hur mycket av hastigheten och rotationen som förändras beror på var och i vilken riktning impulsen appliceras och hur stor massa respektive tröghetsmoment kroppen har. Impulsen som genereras i en kollision beskrivs av ekvation (4.8).

$$j = \frac{-(1+e)v_{rel} \cdot n}{\frac{1}{m^A} + \frac{1}{m^B} + \frac{(r_\perp^A \cdot n)^2}{I^A} + \frac{(r_\perp^B \cdot n)^2}{I^B}} \tag{4.8}$$

där e är restitution (hur mycket av energin som bevaras vid kollisionen), v_1^{rel} är den relativa hastigheten i kollisionspunkten mellan objekt A och B innan kollisionen, m objektens massa, I objektens tröghetsmoment och r vektorn från tyngdpunkten till kollisionspunkten för respektive objekt. Symbolen $\perp$ betyder att det är den ortogonala vektorn som används. Normalen n antas utgå från objekt A. Den beräknade impulsen j appliceras sedan på de båda inblandade objekten.
Ekvation (4.9) och (4.10) beskriver hur de stela kropparnas nya linjära hastighet v' respektive nya rotationshastighet ω' påverkas av impulsen beroende på deras massa m och tröghetsmoment I. För att den relativa hastigheten och rörelsemängden ska stämma efter kollisionen dras impulsens verkan bort från den ena kroppens hastighet medan den läggs till på den andras (eller vice versa, beroende på vilken kropp normalen är riktad mot).

$$v' = v \pm \frac{j}{m} n \tag{4.9}$$

$$\omega' = \omega \pm \frac{r_\perp \cdot jn}{I} \tag{4.10}$$

där v och ω är ett objekts linjära hastighet respektive rotationshastighet före kollisionen. Kodmässigt behövs inga fler variabler för att utföra dessa beräkningar utöver de som redan nämnts i avsnitt 4.2 och 4.3.

4.4.2 Friktion

Då friktion uppstår när två objekt rör varandra måste denna också beräknas och appliceras vid en kollision. Även friktion kan simuleras med hjälp av impulser. För att beräkna friktionsimpulsen används samma formler som för kollisionsimpulsen med undantaget att en friktionskonstant används istället för restitution och normalen roteras 90 grader. Denna beräkning stämmer inte när statisk friktion skall appliceras, dvs när friktionskraften fortfarande är jämnstor med den drivande kraften. Detta löses genom att använda en annan formel när objektens relativa hastighet är under en viss gräns. Under gränsen sätts friktionsimpulsen helt enkelt så att den eliminerar objektets hastighet fullständigt.

4.4.3 Separating Axis Theorem

Separating Axis Theorem är en metod för att detektera kollisioner mellan konvexa polygoner [24]. Metoden går ut på att polygonerna projiceras på olika axlar. På så sätt kan man se om de överlappar för just den axeln. Två polygoner kolliderar om och endast om de överlappar för alla möjliga axlar, där axlarna definieras av normalerna för de båda polygonernas alla kanter.

Så fort en separerande axel hittas, dvs en axel där de inte överlappar, kan kollision uteslutas. Detta medför att SAT är väldigt effektiv när objekt inte kolliderar. Genom att spara föregående uppdaterings separerande axel och testa den först kan algoritmen göras ännu effektivare. När ingen separerande axel hittas används den axel där polygonerna överlappar minst som normal för kollisionen.

4.5 Fluiddynamik

Fluiddynamik är ett område inom fysiken som beskriver beteendet hos icke-fasta kroppar som vätskor och gaser. Inom spelmotorn är fluiddynamik intressant för att grafiskt beskriva vätskor, rök och eld, samt hur de interagerar med sin omgivning. Implementationen av fluider i spelmotorn bygger främst på två ekvationer från Navier-Stokes ekvationer.

4.5.1 Navier-Stokes Ekvationer

Navier-Stokes ekvationer beskriver en vätskas rörelse och dess relation till vätskans egenskaper. Ekvationerna som arbetades fram av Claude-Louis Navier och Sir George Gabriel Stokes i början av 1800-talet är ett av millennieproblemen, vilket innebär att det idag inte finns några effektiva lösningar av ekvationerna. För att kunna använda ekvationerna i praktiken så används förenklingar som är lösningsbara, vilket således även kommer användas för implementationen i spelmotorn.

En fluids rörelse modelleras som ett fält av hastighetsvektorer, där varje punkt i rummet tilldelas en hastighetsvektor. Dessa hastighetsvektorer förändras för varje tidssteg beroende på ett flertal faktorer, t.ex. förändras hastighetsvektorerna för luft beroende på värme, luftdrag o.s.v. Navier-Stokes ekvationer är en exakt beskrivning över hur hastigheterna förändras för ett oändligt litet tidssteg. Ekvation (4.11) ger oss Navier-Stokes ekvation för hastigheter i kompakt vektorform. Ekvationen kan beskrivas som att förändringen av hastigheten beror på de tre termerna i högerledet [18].

$$\frac{\partial u}{\partial t} = -(u \cdot \bigtriangledown)u + \nu \bigtriangledown^2 u + f \tag{4.11}$$

där ν är den kinetiska viskositeten > 0, u är flödeshastigheten och f är krafter som påverkar fluiden.

Simuleringen av fluider som finns i spelmotorn är grafisk och därmed räcker det inte med ett fält av hastighetsvektorer. I en grafisk simulering ska objekt kunna påverkas av hastighetsvektorerna och sedan målas ut på skärmen. Tyngre objekt förflyttas genom att omkringliggande hastighetsvektorer omvandlas till krafter som påverkar objekten, medan lättare objekt som rök vanligtvis följer med hastighetsfältets rörelse. När det gäller rök så skulle ett tillvägagångssätt kunna vara att simulera varje partikel för sig, men det

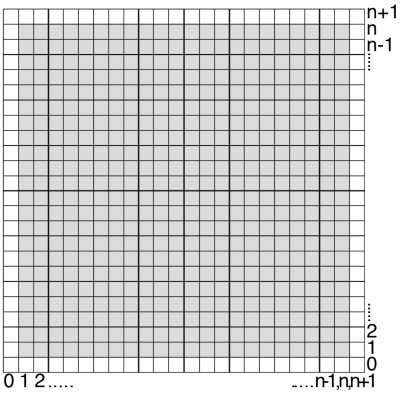
skulle bli mycket resurskrävande. Istället modelleras rök som en distribution av densiteter i ett rutnät. Varje ruta i rutnätet har en viss densitet som beskrivs med ett nummer mellan noll och ett, där noll innebär att det inte finns några partiklar i rutan och ett innebär att rutan är full av partiklar. Densitetens förändring i hastighetsfältet beskrivs av ekvation (4.12) [18].

$$\frac{\partial \rho}{\partial t} = -(u \cdot \bigtriangledown)\rho + \kappa \bigtriangledown^2 \rho + S \tag{4.12}$$

där u är flödeshastigheten, ρ är densiteten, κ är fluidens diffusionskonstant och S är densitet som läggs till genom källor.

4.5.2 Fluid i en låda

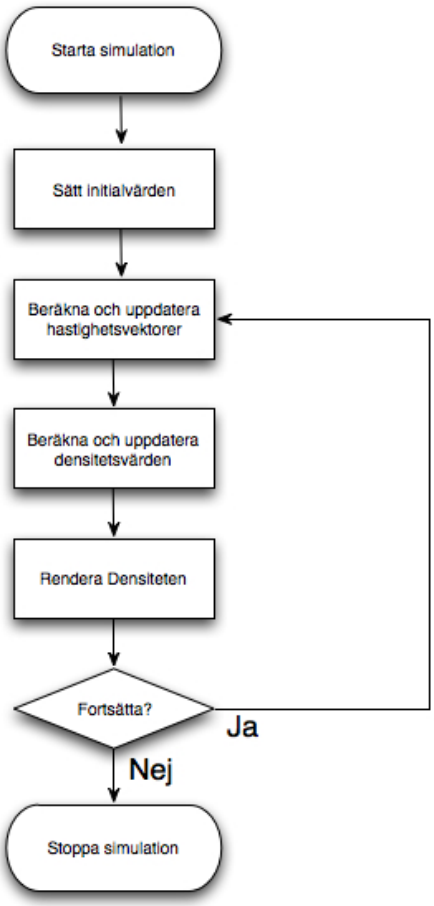
I simuleringen av fluider så används ett rutnät där hastighet och densitet beräknas från mitten av varje ruta. Rutnätet illustreras i figur 4.1 och är NxN stort med ett omgivande lager av rutor där gränsvärdena för fluiden återfinns. Med gränsrutorna har rutnätet en storlek på (N+2)x(N+2) rutor.



Figur 4.1: Rutnätet som används i simuleringen av fluider

4.5.3 Algoritm

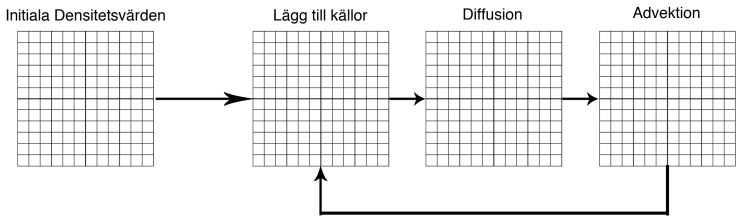
Simuleringen startar genom att varje ruta tilldelas initiala värden för hastighet och densitet. Därefter påbörjas en beräknings- och renderingsrutin där rutnätet uppdateras med nya beräknade värden och densiteten renderas på skärmen. Rutinen beräknar först nya värden för hastighetsvektorerna, därefter nya värden för densiteten som till sist renderas. En grov beskrivning av vår algoritm illustreras i figur 4.2. I nästkommande delkapitel kommer varje steg i algoritmen beskrivas i detalj.



Figur 4.2: En grov översikt över den algoritm som används i simulationen

4.5.4 Densitetsberäknare

Densitetberäknarens uppgift är att räkna ut nya värden på rutnätets densiteter. Om vi återgår till ekvation (4.12) ovanför så ser vi att de nya densiteterna beräknas utifrån tre faktorer på höger sida i ekvationen. Den första faktorn säger att densiteten bör följa hastighetsfältet, den andra säger att röken bör lösas upp i en viss ratio och den tredje säger att densiteten ökar på grund av att rök tillsätts genom källor. Strukturen i vår densitetsberäknare illustreras i figur 4.3 [18].



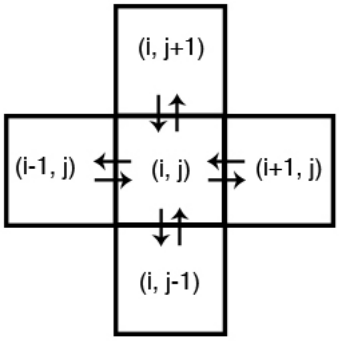
Figur 4.3: Flödesdiagrammet visar den algoritm vi använder i vår densitetsberäknare

Källor

Källor är punkter där vätska och rök läggs till. Exempel på källor i en spelvärd kan vara vatten från en bäck eller rök från en eld. Rutinen itererar igenom rutnätets alla rutor och adderar mängden vätska eller rök som en källa avger i respektive ruta.

Diffusion

Diffusion innebär att densiteten i en ruta sprids ut till närliggande rutor. Det är en spridningsprocess som finns i naturen för att uppnå jämvikt i t.ex. luft och vatten. Diffusion är en egenskap som är olika för varje vätska och gas, således används en variabel, *diff*, som bestämmer diffusionsegenskapen hos ett material. När *diff* är större än noll så sprider sig densiteten i en ruta till närliggande rutor. De nya värdena för densitetsmatrisen som diffusionen ger upphov till beräknas genom att lösa det linjära ekvationssystemet för rutnätet [18].



Figur 4.4: En illustration över den diffusion som sker i densitetsberäknaren

Advektion

Advektion beskriver hur ett ämne transporteras i en fluid som exempelvis sotpartiklar eller sand.

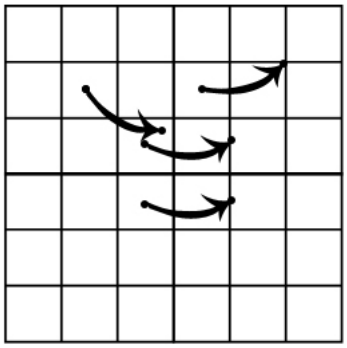
För att beräkna advektionen i simuleringen så modelleras densiteten här istället som ett set med partiklar, vilket innebär att enbart partiklarna genom hastighetsfältet behöver följas för att se hur ämnen transporteras i fluiden [18]. I lösningen antas att mitten av varje cell i rutnätet är en partikel. Men ett problem som uppstår är att partiklarna måste konverteras tillbaka till värdena i vårt rutnät. Istället används en annan metod där vi försöker hitta partiklarna som vid en viss tidpunkt befinner sig i mitten av en cell. Mängden densitet som varje partikel för med sig fås sedan ut genom linjär interpolering av densiteten vid deras startpunkter från de fyra granncellerna.

Lösningen av advektions-steget blir därmed följande: Vi startar med två matriser, en som innehåller densitetsvärdena från föregående tidpunkt och en som kommer innehålla de nya värdena. Sedan genomförs följande procedur:

1. För varje gridcell med de nya värdena följer vi cellens mittpunkt bakåt i tiden genom hastighetsfältet.

2. Linjär interpolering görs utifrån griden med de föregående densitetsvärdena och nya värden tilldelas till matrisen för den nuvarande tidpunkten.

Advektionsproceduren illustreras i figur 4.5.



Figur 4.5: Partiklarna som befinner sig i mitten av en cell följs bakåt i tiden för att beräkna transporten av ämnena i fluiden

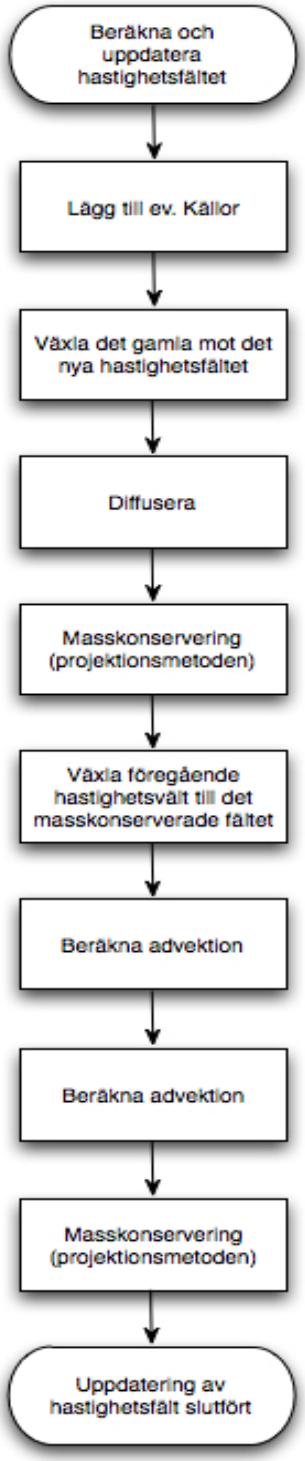
Sammanfogning

Densitetsberäknarens delar sammanfogas till slut i en rutin där källor läggs till, diffusion görs och advektion beräknas.

4.5.5 Hastighetsberäknare

De faktorer som påverkar hastigheten hos fluiderna beror på tre faktorer: tillägg av krafter, viskositet och självadvektion. Viskositet är en egenskap hos fluider som kan betecknas som en fluids tröghet eller inre motstånd, t.ex. har olja högre viskositet än vatten. Självadvektion kan beskrivas som att hastighetsfältet rör sig utmed sig självt.

Om vi återgår till ekvation (4.11) och (4.12) så ser vi att dessa är väldigt lika varandra, således kan flera steg som används i densitetsberäkningen även användas i beräkningen av hastighetsvektorerna. I figur 4.6 visas flödesdiagrammet som beskriver algoritmen som används när hasighetsfältets nya värden beräknas och uppdateras.



Figur 4.6: Flödesdiagrammet visar den algoritm som används i hastighetsberäknaren

Masskonservering - Projektionsmetoden

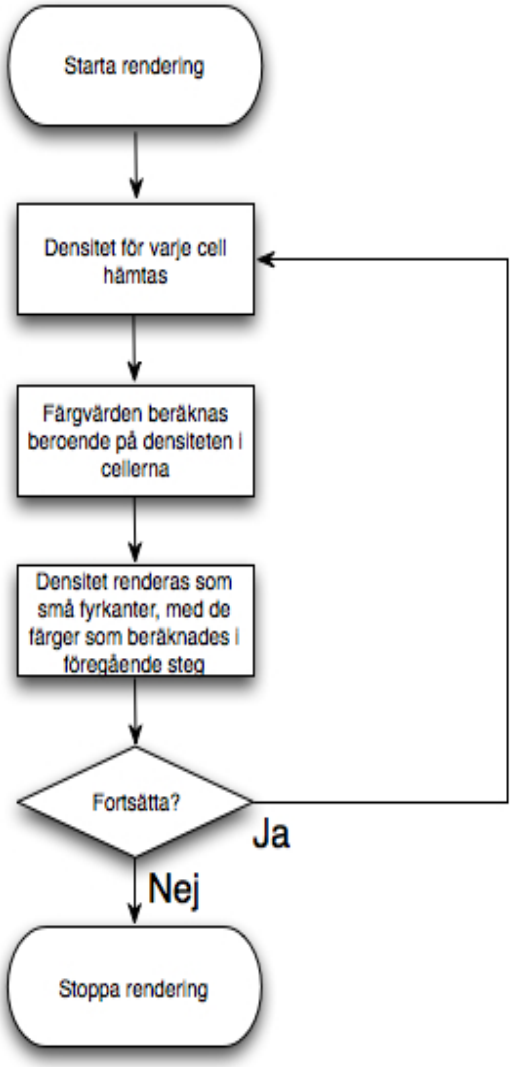
Vid beräkningen och uppdateringen av hastighetsvektorerna används en teknik som inom fluiddynamik kallas för projektionsmetoden. Denna teknik är en effektiv numerisk metod för att lösa tidsberoende problem som involverar inkompressibla fluider, som till exempel gaser. Projektionsmetoden bygger på ett koncept från matematiken kallat Helmholtz-hodge dekomposition [18], vilken säger att ett vektorfält kan brytas upp i ett solenoidalt fält och ett konservativt fält. Ett solenoidalt fält är ett vektorfält vars divergens är definierad och lika med noll och ett konservativt fält är ett vektorfält vars rotation är definierad och lika med noll.

Gränsvärden

I fluidberäkningarna ska fluiden vara "instängd" och finnas kvar inom rutnätet så att densiteten inte kan försvinna ut genom kanterna. För att åstadkomma detta användes rutor i kanterna av fluidlådan, vilket illustrerades i figur 4.1. Dessa rutor behandlas i en speciell rutin som ser till att fluidens hastighet ut från lådan sätts till noll [18].

4.5.6 Rendering

Rendering av fluiden som i det här fallet är en gas sker med hjälp av OpenGL. Gasen renderas genom en rutin som hämtar densiteten för varje cell i vårt rutnät, skapar färger beroende på densiteten och som sedan målar ut gasen som många små fyrkanter. Algoritmen för rendering av gasen illustreras i figur 4.7.



Figur 4.7: Flödesdiagrammet visar en algoritm för rendering av gas

Beroende på vilken typ av gas som renderas så kan steget där färgen för fyrkanterna beräknas ändras för att anpassas efter gasens färg och egenskaper, t.ex. kan rök från eld renderas som grå färg. Färgen som beräknas beror dessutom på densiteten, d.v.s. mängden gas i varje cell i rutnätet, det betyder att gasen är mer framträdande och tydlig där densiteten är hög.

4.5.7 Utökning och optimering av algoritmerna

Beräkning och rendering av gas kräver relativt mycket prestanda från enheten som mjukvaran körs på. När vi testade på iPhone lyckades vi komma upp

i 9 FPS, vilket inte är godtagbart i ett spel. Däremot går det att optimera algoritmen genom att till exempel minska noggrannheten i beräkningarna och minska antalet rutor i rutnätet. Inom ramarna för projektet har optimering av kod och algoritmer inte hunnits med, vi lämnar det således för framtida arbete.

Algoritmerna som har beskrivits i kapitlet är utbyggbara och kan utökas för att stödja vatten eller andra fluider. Det är också något som tyvärr inte hunnits med i projektet och vi lämnar således även detta till framtida arbete.

Kapitel 5

Utförande

Följande kapitel beskriver det arbete och de val som har gjorts under projektets gång. Det är främst uppdelat i de huvudområden som spelmotorn består av, men också delar som beskriver arbetet gemensamt för alla områden. Delarna är skrivna i kronologisk ordning.

5.1 Litteraturstudie

Att hitta och sortera ut relevant litteratur för att utveckla en spelmotor har varit en stor och lärorik del av projektet. Det finns väldigt mycket litteratur av olika vetenskapliga naturer att tillgå och ofta är det stora skillnader rent faktamässigt mellan olika källor. Vissa delar, t.ex. hur man ska hitta och lösa kollisioner i en fysikmotor är fortfarande ett område för forskning. Detta har lett till att det ibland har vart svårt att få ta del av de absolut senaste resultaten och framstegen inom området, men för övrigt är det bara positivt att seriös forskning bedrivs.

När litteraturstudien började var det svårt att direkt gå in på avancerade vetenskapliga artiklar. Vi fick börja från grunden och böcker som *Beginning iPhone SDK: Programming with Objective-C* [7], *Vägen till C* [15], *Linjär algebra* [14] och *Game physics engine development* [6] var till stor hjälp för att komma igång. Större delen av gruppen hade precis läst kursen *Computer graphics* vilket var till stor hjälp för att förstå och komma igång med grunderna inom grafiken, mer precist OpenGL.

När grunderna var någorlunda utforskade var det dags att undersöka mer avancerad litteratur. Fysiken och grafiken blev ett naturligt fokus under utvecklingen och därför också ett område för djupare litteraturstudier. Genom att kombinera fakta från olika källor kunde en gemensam bas för den viktigaste teorin byggas upp. Självklart har denna bas utvecklats under projektets gång precis som koden har utvecklats och skrivits om. Den viktigaste teorin beskrivs närmare i kapitel 4.

5.2 Definition av spelmotorn

Begreppet spelmotor myntades någon gång i mitten av 90-talet. Det växte fram samtidigt som de populära spelen Doom och Quake släpptes. Många andra utvecklare valde att licensiera och använda kärnorna i dessa spel och sedan designa egen grafik och eget innehåll när ett nytt spel skulle utvecklas istället för att börja om från början. Idag utvecklas nästan alla spel och spelmotorer helt skilt ifrån varandra. Dagens spelmotorer är en samling komponenter som tillsammans med grafik och innehåll såsom spelvärldar, karaktärer och spelregler utgör ett spel.

5.2.1 Struktur och funktionalitet

När utvecklingen av spelmotorn skulle börja i projektet var första steget att definiera vilka komponenter den skulle innehålla och vad för funktionalitet dessa komponenter tillsammans skulle ha. Spelmotorn delades upp i fyra huvudkomponenter:

- *Fysikmotor*
  Simulerar fysiken i spelvärlden. Ska bland annat uppdatera ett objekts position, hastighet och rotation. Fysikmotorn hanterar även kollisioner mellan objekt.
- *Grafikmotor*
  Har hand om den visuella representationen av spelvärlden. Kan rita ut godtyckliga polygoner i 2D med texturer och färger. Kan också rita ut ljuskällor som medför att kringliggande polygoner avger skuggor.
- *Ljudmotor*
  Hanterar alla ljud i spelvärlden. Kan spela upp ljud associerade med en viss händelse men också bakgrundsljud.
- *Interaktion*
  Ger olika möjligheter för en användare att påverka spelvärlden, bland annat genom att ta in data från accelerometern och multi-touch-skärmen.

Utöver huvudkomponenterna finns också en del som kopplar ihop dem så att de tillsammans bildar en spelmotor.

5.2.2 Portabilitet

När spelmotorn definierades hade alla i gruppen ett gemensamt mål att motorn skulle vara så portabel och plattformsoberoende som möjligt, även om spelmotorn i början bara utvecklades för iOS. Det var framförallt fysikmotorn vi ville hålla portabel och därför togs beslutet att den skulle skrivas i C. Viljan att hålla spelmotorn portabel speglades sedan även i beslutet att skriva grafikmotorn i C. Detta beskrivs närmare i avsnitt 5.4.6.

5.2.3 Gränssnitt

En del av definitionen var att spelmotorn skulle vara lätt att använda. Tanken var att ett enklare skriptspråk skulle utvecklas och kunna användas för att konstruera spelvärlden och bestämma dess innehåll och egenskaper. Utöver skriptspråket så skulle också ett lättanvänt API implementeras.

5.3 Fysik

Fysiken har haft en central roll under hela utvecklingen av spelmotorn och har genomgått ett antal viktiga stadier där funktionaliteten och gruppens förståelse har ökat för varje steg. Fysikmotorn har gått från att simulera enkla geometriska objekt med förutbestämda beteenden till godtyckligt formade polygoner som roterar och kolliderar.

5.3.1 Beteenden

Utvecklingen av fysikmotorn började med att en beteendemodell skapades. I beteendemodellen fanns funktioner som modifierade ett grafiskt objekts position, skala och rotation. I fortsättning kommer dessa funktioner att refereras till som *beteenden*. Varje objekt associeras med ett beteende som anropas vid varje skärmuppdatering och modifierar objektet innan det ritas ut. Detta var en bra start för att se hur objekten rörde sig på skärmen och för att bekanta sig med skärmens koordinater.

Det första fysikaliska beteendet kom i och med att vi utvecklade en datatyp för partiklar med tillhörande funktioner för att modifiera dem. Detaljerna för partikelns rörelser beskrivs närmare i avsnitt 4.2. En partikel kopplades ihop med ett grafiskt objekt genom ett spelobjekt och på så sätt kunde enkla linjära rörelser simuleras. Det grafiska objektet och den underliggande partikeln hade fortfarande var sin position och därför behövde de synkroniseras med hjälp av beteendemodellen. Ett fysikaliskt beteende utvecklades som dels synkroniserade det grafiska objektet med partikeln men också applicerade de krafter som i varje uppdatering medför att partikelns hastighet och position integreras som följd av den genererade accelerationen. I och med att partiklar inte har någon utsträckning kan de inte användas för att simulera rotationer och representerar inte det grafiska objektet fullständigt.

5.3.2 Stela kroppar

För att kunna simulera avancerad fysik med rotationer och kollisioner finns det två vägar att gå:

- *Sammansatta partiklar*
  Varje objekt är uppbyggt av flera partiklar som är sammankopplade

med varierande styrka. En partikel roterar inte i sig men alla partiklars kombinerade linjära rörelser kan simulera rotation för hela objektet.

- *Stela kroppar*
  Ett objekt består av en stel kropp med en viss utsträckning och form. De linjära rörelserna är identiska med en partikels, men "riktiga" rotationsrörelser simuleras också. Även de stela kropparna kan kopplas ihop för att forma avancerade objekt men det är inte nödvändigt för enkla geometriska former och polygoner. Mer om stela kroppars rörelser i avsnitt 4.3.

Båda vägarna är intressanta och används i existerande fysikmotorer, men på senare år har stela kroppar varit valet i de flesta fall. Detta beror mycket på att de ger en mer korrekt beskrivning av verkligheten vilket leder till att man kan använda mer generella och robusta metoder för att hantera kollisioner [6]. Sammansatta partiklar var lockande till en början, dels för att fungerande partiklar redan hade implementerats men också för att det verkade relativt enkelt gentemot stela kroppar. Efter att ha läst Chris Hecker's artikelserie [9][10][11], undersökt och testat simuleringarna på *My Physics Lab* [12] och sakta men säkert börjat förstå matematiken bakom rotationer (se avsnitt 4.3) blev övertygelsen stor att stela kroppar var rätt väg att gå.

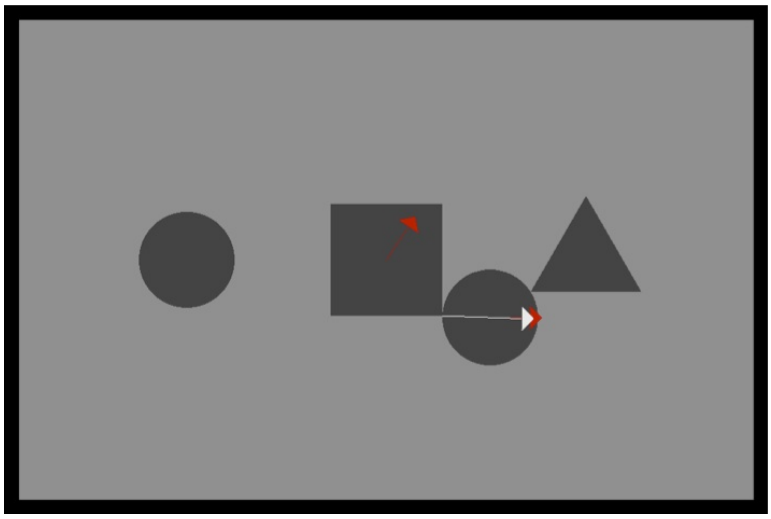
I och med att viss del av partiklars rörelser är densamma för stela kroppar gick mycket av koden för partiklar att använda direkt för stela kroppar. Datatypen och funktionerna utökades med variabler och funktionalitet för rotation och utsträckning. Fysikmotorn hade nu nått ett stadie där den hade fullständig funktionalitet för att simulera ett objekts rörelse i 2 dimensioner. Då objekt inte påverkade varandra kändes simuleringen fortfarande inte speciellt verklighetstrogen. Därför blev nästa steg i utvecklingen just att få objekt att påverka varandra – *kollisioner*.

5.3.3 Enkla geometriska figurer

Första steget för att på ett smidigt sätt kunna testa implementationer av fysikaliska egenskaper var att implementera enkla geometriska figurer som nedan i fig. 5.1. Uppbyggnaden av trianglar och rektanglar gjordes m.h.a. linjer istället för det kanske mer naturliga sättet med en punkt för varje hörn. Anledningen till detta tillvägagångssätt var att funktioner för att kolla om en punkt ligger i en figur kunde reduceras till endast tre och fyra jämförelser för trianglar respektive rektanglar. Dessa funktioner måste hållas så kostnadseffektiva som möjligt för att de inte bara körs flera gånger för varje objekt i världen utan även minst 60 gånger per sekund (1 gång per 0.0167 sekund).

Kollisionsdetektionen för geometriska figurer inspirerades till stor del av implementationen i *Game physics engine development* [6] som handlar om att implementera en fysikmotor från grunden i C++. Boken diskuterar i stor

omfattning olika lösningsmetoder och förklarar varför och när man bör välja vardera.



Figur 5.1: Debugläge med shapes och kollisionsnormaler

Målet för kollisionshanteringen var att hålla den så generell som möjligt. Kollisionshantering består i stort sett av att upptäcka kollisioner (“detektor”) och att lösa kollisioner (“beräknare”/“resolver”). För att slippa särskilja de geometriska figurerna när de skickades in till detektorn så skapades polymorfism i C. Detta finns inte inbyggt i språket som standard men går att uppnå genom att använda sig av strukturer som börjar med samma typ av element. Detta element används för att determinera vilken typ strukturen egentligen har och består i detta fall av en enumererad typ (Enum). På detta sätt kunde kollisionsfunktionerna hållas väldigt generella. Funktionen tar emot en generell geometrisk figur och avgör sedan vad det är för typ och hur den ska behandlas. För att upptäcka och lösa kollisioner används till stor del det omfattande vektor-biblioteket (4.1.2) som har implementerats och utökats löpande under hela utvecklingsperioden för att täcka alla behov.

5.3.4 Polygoner

Som tidigare nämnts var implementationen av geometriska figurer och kollisioner mellan dessa endast det första steget för att i ett tidigt skede kunna upptäcka felaktiga implementationer av Newtonsk fysik samt vektor- och matrismatematik.

För att kunna skapa verklighetstrogna miljöer i spelmotorn så måste man kunna använda sig av andra former än t.ex. rektanglar och cirklar. För att tillgodose detta implementerades en struktur för generella polygoner. Dessa polygoner kunde inledningsvis vara såväl konkava polygoner som konvexa

polygoner med ett godtyckligt antal hörn. De konkava polygonerna togs vid ett senare skede bort, mer om detta i 5.3.6. För att kunna rendera sådana polygoner krävdes en funktion som kunde dela upp alla sorters polygoner i trianglar, med anledningen att endast trianglar kan renderas till skärmen (mer om detta i avsnitt 5.4.3).

5.3.5 Optimering

Som tidigare nämnts utvecklades till att börja med kollisionsdetektionen i fysikmotorn på ett sådant sätt att den var anpassad att fungera generellt för alla primitiva geometriska figurer. Allt eftersom att migration från geometriska figurer till generella polygoner skedde så har detta skrivits om till en helt ny detektor. Så länge de primitiva figurerna användes fanns det en metod för vardera figur mot vardera utav de andra figurerna, vilken av dessa metoder som skulle användas valdes i sin tur av en funktion som tog in två godtyckliga figurer. Kodmässigt var detta väldigt omfattande men det gav en pixel-perfekt kollisionsdetektion för figurerna. När detta under projektets utveckling skrevs om så anpassades kollisionsdetektionen till att använda sig av *Faster line segment intersection* [16] och kodmässigt blev det väldigt reducerat i förhållande till den första detektorn. Detta p.g.a. att i princip hela den pixel-perfekta detektionen bestod av en funktion som kontrollerade alla kanter i den ena polygonen mot kanterna i den andra polygonen och om någon av dessa överlappade med varandra.

Den nya detektorn kompletterades även med ett lager som körs innan den pixel-perfekta detektionen. Detta lager använder sig av bounding volumes och kontrollerar m.h.a. dessa om polygonerna är tillräckligt nära varandra för att kunna innehålla överlappande kanter. På så sätt minimeras antalet gånger som den mest tidskrävande delen måste köras, d.v.s. den pixel-perfekta detektionen. Som bounding volumes används cirklar eftersom dessa ger väldigt lätta jämförelser för att kontrollera överlappning, endast summan av radierna samt avståndet mellan mittpunkterna på de omslutande cirklarna måste jämföras.

5.3.6 SAT

Genom att använda *Faster line segment intersection* [16] kunde mängden kod reduceras och kollisionsdetektion göras generell för konkava och konvexa polygoner. Tyvärr medförde det också många specialfall där kontaktdata var mycket komplicerad att beräkna. En annan effektiv metod för att detektera kollisioner är Separating Axis Theorem (SAT, se avsnitt 4.4.3). SAT fungerar endast för konvexa polygoner, men förutom effektivitet medför det också att kontaktdata kan beräknas relativt enkelt. På grund av de komplikationer som uppkom av att detektionen skulle kunna hantera konkava polygoner beslutades det att dessa skulle överges. Konkava polygoner kan hur som

helst skapas genom att kombinera konvexa polygoner. SAT implementerades och för första gången var kollisionsdetektionen exakt utan att speciallfall behövde hanteras.

5.3.7 Kollisionsberäkning

Kollisionsberäkning eller "Collision resolving" har tillsammans med kollisiondetektion varit en av de större utmaningarna i projektet. Detta beror dels på den avancerade matematiken bakom kollisioner men också på att det finns väldigt många olika lösningar. Att de flesta existerande lösningar hanterar 3D istället för 2D har inte gjort det hela lättare, då det inte alltid är lätt att reducera dem till 2 dimensioner. Arbetssättet har varit att läsa och försöka förstå så många lösningar som möjligt. Detta har också inneburit att koden har fått sig en rejäl uppdatering flertalet gånger. En detalj som var klar i ett tidigt stadie som följd av hur resten av fysikmotorn utvecklades var att impulser måste användas istället för krafter. Det grundläggande flödet i kollisionsberäknaren och vilken data som behövdes från detektorn var också tvunget att definieras relativt tidigt för att samtidigt kunna fortsätta utvecklingen av detektorn.

Den huvudsakliga uppgiften för kollisionsberäknaren är att räkna ut och applicera förändringen i hastighet på två objekt som kolliderar. Fysiken bakom vår lösning beskrivs detaljerat i avsnitt 4.4.

5.3.8 Penetrering

Förutom att räkna ut och förändra hastigheten på objekten efter en kollision har kollisionsberäknaren också en annan uppgift. I vår implementering detekteras inte en kollision förrän objekten i fråga överlappar eller penetrerar varandra. Detta beteende stämmer naturligtvis inte ihop med verkligheten och för att detta inte ska synas måste penetreringen beräknas tillsammans med hastigheten. Det enklaste sättet att räkna ut penetrering är helt enkelt att linjärt flytta objekten precis så mycket så att de inte längre penetrerar varandra. Att linjärt flytta ett objekt som kan rotera kan i vissa lägen se väldigt konstigt ut [6]. Vi valde därför att lösa penetrering genom både linjär förflyttning och rotation. Hur mycket av penetreringen som uträknas genom vardera beror på kontaktpunktens position i förhållande till objektets tyngdpunkt, objektets massa och tröghetsmoment.

Iterativ lösning av penetrering

Nackdelen med att lösa penetrering på detta sätt är att om fler objekt är involverade i kollisionen så kan lösningen av penetreringen i en kontaktpunkt leda till att penetreringen i en annan förvärras. Detta syns genom att objekt upplevs att vibrera och glida in i varandra. För att undvika detta kan en iterativ lösning av penetrering användas. Det innebär att man räknar ut

mindre delar av penetreringen i alla kontaktpunkter iterativt och uppdaterar alla andra kontakter efter varje förflyttning. Varje kontakt måste självklart beräknas minst en gång och om man ökar antalet iterationer förbättras resultatet märkbart.

5.3.9 Vilande objekt

I och med att impulser används istället för krafter för att förändra hastigheten på två objekt när de kolliderar försvinner möjligheten att ta nytta av faktumet att krafter kan eliminera varandra. Det medför att objekt som vilar på varandra egentligen inte är i vila utan för varje uppdatering åker de in i varandra och en kollision detekteras och uträknas. Dessa *mikrokollisioner* kan ge också upphov till att objekt upplevs som vibrerande. Det finns ett par metoder för att dölja dessa mikrokollisioner:

- När den relativa hastigheten mellan två objekt är lägre än en viss gräns begränsas impulsen så att den endast tar ut den relativa hastigheten genom att sätta restitutionen till noll.
- När en kollision detekteras tas hastighetsförändringen som uppkom ur accelerationen i föregående uppdatering bort.

Båda dessa metoder används för att dölja mikrokollisioner.

5.3.10 Spekulativa kontakter

En alternativ metod att lösa kollisioner är spekulativa kontakter. Genom att upptäcka kollisioner innan de verkligen äger rum kan t.ex. en stor del av penetrationen mellan objekt undvikas och generellt bidra till ett stabilare kollisionssystem. Då tiden inte räckte till för att implementera spekulativa kontakter beskrivs det närmare i avsnitt 8.5 under framtida arbete.

5.3.11 Matriser

I fysikmotorn användes till en början en vektor och ett flyttal för att lagra ett objekts position respektive orientation. Då ett objekt ska ritas ut måste en matris med dessa som grund konstrueras för att grafikmotorn och slutligen OpenGL ska kunna transformera koordinaterna för objektet, så det kan ritas ut med position och rotation som stämmer överens med det fysikaliska objektet. För att underlätta denna interaktion mellan fysik- och grafikmotor ändrades representationen i fysikmotorn till en matris som innehåller både ett objekts position och rotation. Matrisen kan sedan skickas direkt till grafikmotorn. För att slippa skicka matrisen till grafikmotorn och minska minnesanvändningen så sparas matrisen endast på en plats i minnet som ägs av fysikobjektet och refereras sedan direkt i den grafiska representationen.

5.3.12 Hantering av krafter

Så här långt hanterade fortfarande beteendemodellen vilka krafter som skulle appliceras för ett objekt. Detta krävde att ett beteende var definierat för alla kombinationer av krafter som kunde verka på ett objekt, vilket inte är speciellt dynamiskt och skalbart. En mer generell lösning för att hålla reda på vilka krafter som ska appliceras på ett objekt är *kraftgeneratorer*. Det blev därför en naturlig och nödvändig utökning av fysikmotorn. Kraftgeneratorer beskrivs närmare i avsnitt 4.2.2. Beteendemodellen förändrades men behölls i viss mån för att fortfarande ge stöd för förutbestämda rörelser.

5.4 Grafik

I följande avsnitt kommer implementation av grafik gås igenom. Avsnittet kommer beskriva implementationen av grafiska effekter som ljus och skuggor, samt shaders, texturer och triangulisering av polygoner.

5.4.1 Exempel från Apple

När utvecklingen började var det ingen i gruppen som hade någon större erfarenhet av utveckling med OpenGL. Apple tillhandahåller en enkel exempelapplikation som använder OpenGL där en kvadrat ritas ut och förflyttas upp och ned. Det blev naturligt att testa och förstå exempelapplikationen för att sedan utöka den efter våra behov. Exemplet innehåller den grundläggande klassen EAGLView som sätter upp nödvändiga delar av OpenGL och kopplar ihop det med resten av applikationen. All rendering sker i en och samma funktion och all data skapas statiskt i funktionen. För att dynamiskt kunna lägga till objekt att rendera och ge dem olika färger och texturer behövdes därför en omstrukturering. Ett val gjordes som innebar att EAGLView skulle behållas i sitt ursprungliga tillstånd på grund av dess grundläggande natur. Renderingskoden flyttades till olika funktioner i en renderingsmodell, “ViewModel”, vilket liknande beteendemodellen i fysikmotorn som diskuteras i avsnitt 5.3.1. På så sätt kunde olika typer av rendering separeras, vilket senare blev väldigt användbart när ett renderingsläge där fysik och kollisioner testas skulle implementeras. I renderings-modellen skapades bland annat olika funktioner för att rita ut med färger och texturer men också för att ladda in den data som behövs för att rendera en bildruta. Apples exempel innehåller renderingskod för både OpenGL ES 1.0 och 2.0. Under omstruktureringen fokuserades utvecklingen framförallt på ES 2.0 men för att ge stöd för äldre enheter behölls också koden för ES 1.0.

5.4.2 Shapes - test av fysik och kollisioner

För att kunna testa om diverse fysikberäkningar och kollisioner fungerade på rätt sätt genomfördes tester grafiskt med de olika formerna. De första for-

merna som testades var cirklar, rektanglar och trianglar. Vad som undersöktes var hur formerna rörde sig (förflyttning och rotationer) vid påverkan av olika krafter, t.ex. gravitation och kollisionskrafter, samt om det fanns några buggar i kollisionsdetektionen/kollisionsresolvern. Ett antal buggar upptäcktes och en närmare genomgång av koden utefter testresultaten genomfördes. Bl.a. kunde en cirkel glida igenom både en triangel och en kvadrat om den kolliderade ifrån en viss vinkel, och vid vissa tillfällen fick objekten så hög hastighet vid en kollision att de åkte utanför skärmen.

5.4.3 Triangulisering av polygoner

En funktion för att dela upp alla generella polygoner i enbart trianglar utan att lägga till nya hörn har utvecklats. Anledningen till att vi var tvungna att skriva denna funktion var att OpenGL inte stödjer rendering av andra former än trianglar (bortsett från linjer). Från början gjordes försök att dela upp generella polygoner i trianglar som låg strukturerade på ett visst sätt, *triangle-strips*. Det innebär att trianglarna hänger ihop så man slipper ange de hörn som tillhör föregående triangel igen. Av denna anledning reduceras mängden data som måste skickas för rendering, vilket är positivt eftersom det tar upp mindre grafikminne och det krävs mindre antal grafiska beräkningar. Detta i sin tur leder till att fler objekt kan renderas per skärmuppdatering.

Tyvärr lyckades inte försöken med triangle-strips. Vi blev istället tvungna att nöja oss med att dela upp polygonerna i trianglar som inte hängde samman på detta sätt. Anledningen till att detta misslyckades är att det inte är möjligt att dela upp alla olika konkava polygoner i triangle-strips utan att fler hörn behöver läggas till inuti polygonens kropp. Faktum är att konkava polygoner ställer till det ganska mycket i triangulisering. För att kunna triangulisera en konkav polygon skrevs en funktion som kunde bestämma om en kant var inom en polygon eller utanför polygonen, utan att ha figuren färdig att kollisionstesta mot. Genom att läsa på om *Linjär Algebra* [14] lyckades vi åstadkomma en smidig implementation m.h.a. normaler och skalärprodukter mellan dessa.

5.4.4 Texturer

Ett önskemål för att kunna få fler möjligheter att ändra utseendet på spel som använder sig av spelmotorn var att kunna använda texturer på objekt.

Därför implementerades en klass för att ladda in texturer kallad “GLTexture”. Stöd för texturer i OpenGL ES 1.0 och 2.0 var även tvunget att läggas in. Utöver det så behövdes det läggas till koordinater som talar om hur texturen ska placeras på objekten. För att testa om texturer fungerade utfördes tester genom att ladda in en godtycklig bild som en textur och placera denna på en rektangulär figur. Detta misslyckades, vilket efter yt-

terligare efterforskning visade sig bero på att bilden som texturerna laddas in från var tvungen att vara kvadratisk och vara minst i dimensionen 64x64 pixlar. Bilden beskärdes därför till dimensionen 128x128 pixlar och ett nytt försök gjordes. Denna gången fungerade det och rektangeln hade fått ett nytt trevligt utseende.

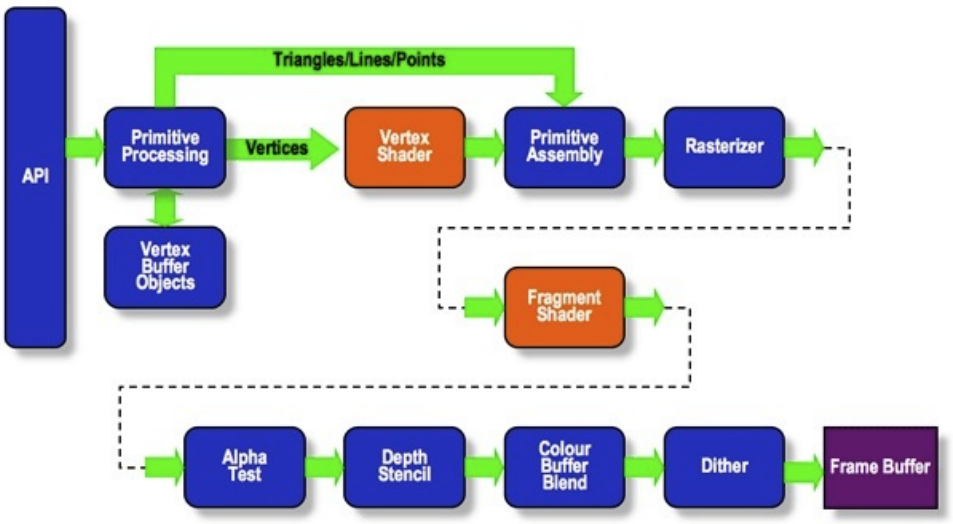
För att kunna återanvända texturer som redan har laddats in skapades en klass kallad "GLTextureManager". Denna klass läser in texturer genom "GLTexture" och sparar dessa i en lista. För att använda en textur skickas namnet på texturen och sedan kontrolleras det ifall texturen redan finns inläst i listan. Om den inte finns så skapas den och läggs till i listan och returneras sedan. I annat fall returneras den endast.

5.4.5 Shaders

För att kunna använda sig av OpenGL ES 2.0 måste man programmera program som byggs upp av så kallade "Shaders". Anledningen till detta är att OpenGL ES 2.0 består av en programmerbar grafisk pipeline, till skillnad från OpenGL ES 1.1, där pipelinen är fixt. En bild på OpenGL ES 2.0 pipeline återfinns nedan i fig. 5.4.5. Ett program består av minst en "Vertex Shader" och en "Fragment Shader".

För en triangel så körs "Vertex Shader" på varje hörn i triangeln och "Fragment Shader" för varje punkt som är innesluten i triangeln. "Vertex Shader" har i huvudsakligt syfte att räkna ut vilken position som hörnen ska ha när de ritas ut medan "Fragment Shader" bestämmer vilka färger som ska användas i den inneslutna arean.

"Vertex Shadern" utför oftast matrisoperationer på vertices, och då denna inte körs så ofta (3 gånger för en triangel) så kan mycket uträkningar utföras. "Fragment Shader" används för att sätta textur och räkna ut ljus på objekt i världen och körs fler gånger ju större yta som täcks, vilket begränsar antalet uträkningar som kan utföras.



Figur 5.2: OpenGL ES 2.0 Grafiska pipeline

5.4.6 Omskrivning till C

Som tidigare nämnts i avsnitt 5.3.1 var ett fysikaliskt objekt från början separerat från det grafiska och de behövde därför synkroniseras innan rendering. Ett behov för en starkare koppling mellan de två objekten växte ganska snart fram. Problemet var att fysikmotorn var skriven i C och grafiken huvudsakligen i Objective-C. Det är i och för sig inga problem att använda C i de delar av programmet som är skrivna i Objective-C. Däremot hade spelmotorns och framförallt fysikmotorns portabilitet varit ett fundament sedan utvecklingens början. Vi ville därför inte skriva fysikmotorn i Objective-C eller koppla den för hårt mot den plattforms-specifika koden. Slutsatsen blev enkel – grafikmotorn skulle konverteras till C.

När grafikmotorn ändå skulle skrivas om togs tillfället i akt och hela representationen av grafikdatan tänktes om.

5.4.7 Optimeringar av grafikdata

Ett Vertex Buffer Object (VBO) är en plats i minnet där man kan läsa in olika typer av grafikdata, t.ex. punkt-, färg- eller texturdata. Datan behöver bara kopieras till grafikminnet en gång, därefter behöver man bara hålla koll på pekaren till VBO:n. Detta innebär att man slipper ladda in grafikdatan vid varje skärmuppdatering, istället använder man ett OpenGL-anrop som säger vart datan ligger i grafikminnet med hjälp av VBO-pekaren. För att underlätta användningen av VBO:s finns ännu ett inkapslingsobjekt – Vertex

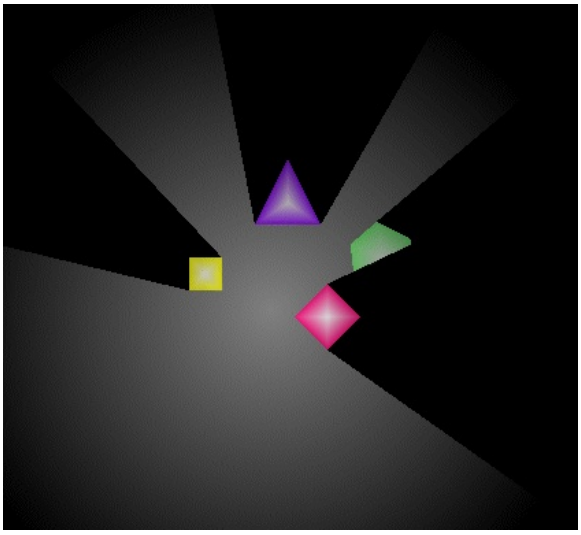
Array Object (VAO). Flera VBO:s kan bindas till samma VAO, t.ex. en för punkter och en för texturkoordinater. När dessa ska användas behöver man bara använda ett OpenGL-anrop för att berätta vilken VAO som skall användas. Den nya representationen av grafikdatan möjliggör användning av både VBO och VAO men fungerar också bra utan dem.

5.4.8 Ljuseffekter

Förhoppningen var att uppnå något i stil med bilden i fig. 5.4.8 nedan. En "per pixel ljuskälla" implementerades under utvecklingens gång men visade sig vara ineffektiv och uppnådde endast 14 FPS på iPhone 4, vilket bedömdes vara för långsamt för att vara användbart. "Per pixel" innebär att för varje punkt på skärmen så räknas ett ljusvärde ut beroende på vad för typ av ljus som laddats.

Istället används i prototyp 2 en textur som beskriver ljusmiljön, en teknik som kallas lightmaps. Texturen genereras genom att ljus renderas till en textur vid programmets start eller innan och sedan laddas till minnet. En position i spelvärlden transformeras till en position i texturen och mängden ljus där bestämmer hur objektet ska belysas. Denna teknik visade sig vara effektivare och mer lämplig för plattformen.

Skuggor var menat att implementeras med hjälp av en liknande teknik kallat shadow maps. Det innebär att skuggorna renderas till en textur och ett värde läses ur texturen som bestämmer om ett objekt ligger i skugga eller ej. Skillnaderna mot lightmaps är dock rätt stora eftersom shadowmaps måste genereras varje gång objekt flyttas i världen och därför har det inte hunnits med att implementeras på ett tillfredsställande sätt. Tekniken beskrivs i Cascaded Shadow Maps av Dimitrov, R. (2007) [27].



Figur 5.3: Ljus- och skuggeffekter som förväntas uppnås

5.5 Interaktion

5.5.1 Val av kontroller

Eftersom både multi-touch och accelerometer finns tillgängliga i alla enheter som har tillgång till App Store så är dessa ganska givna val om man vill att produkten ska nå ut till så många användare som möjligt. Gyroskop finns som sagt endast i de nyaste modellerna och en implementation som använder denna undviks därför. Simulering av ett gyroskop med kompass är varken speciellt precist eller tillgängligt för alla enheter och denna utesluts därför också.

Fördelen med att använda GPS är att det aktiverar användaren. Nackdelarna är dels att användarna inte alltid har tid att förflytta sig till olika platser och dels att GPS-signalen kan variera mycket beroende på vilken enhet och vilket iOS användaren har installerat.

5.5.2 Styra gravitationen med accelerometern

En av grundidéerna till spelkonceptet, som vi hade från början, är möjligheten att påverka spelmiljön genom att vrida på telefonen, alltså genom att avläsa data från accelerometern.

Till en början avlästes endast accelerometerns data och skrevs ut i konsolen för att dels testa precisionen av hårdvaran, men också för att veta hur representationen av datan såg ut. Precisionen visade sig vara ytterst bra och datan som utvanns kunde omvandlas till att representera en punkt i 2D-rummet i spelvärden, med X- och Y-koordinater. Tillsammans med

origo, alltså skärmens mittpunkt, bildade punkten en vektor. En kraft placerades sedan på rörliga objekt med storleken och riktningen av vektorn som startade i objektens mittpunkter. När vektorn var utvunnen och normaliserad så gjordes ett grafiskt test där vektorn ritades ut på skärmen som en pil med startpunkten i origo, se den röda pilen i mitten av fig. 5.1 i kapitel 5.3.3.

5.5.3 Interagera med objekt genom multi-touch

När användaren trycker någonstans på skärmen skapas ett event och en metod kallad "Touches Began" körs. Om denna metoden inte är implementerad så händer det ingenting när man klickar på displayen. Utöver "Touches Began" finns även två andra metoder för att hantera touch-event som heter "Touches Moved" respektive "Touches Ended". Dessa två körs då användaren flyttar en beröring utmed skärmen och då en beröring avlägsnas från skärmen.

Den första implementationen av touch kunde endast hantera fallet då en tryckning på skärmen var närvarande. Tryckningen översattes till en punkt med X- och Y-koordinater som skrevs ut i konsollen. Utskrift skedde vid första tryckningen, när den förflyttades samt när den togs bort. Utifrån detta insågs det att X- och Y-koordinaterna representerade en pixel med (X, Y) = (0, 0) i skärmens övre vänsterkant. Utöver den observationen så framgick även att "Touches Began" och "Touches Ended" endast körs en gång, medan "Touches Moved" körs varje gång beröringen förflyttar sig från en pixel-representation till en annan. Eftersom X och Y representerade en pixel var dessa tvungna att omvandlas till en punkt i spelrymden. Detta skedde enligt ekvationerna 5.1 och 5.2 nedan.

$$x' = \frac{(x - (w/2))}{50} \tag{5.1}$$

$$y' = -\frac{(y - (h/2))}{50} \tag{5.2}$$

där x' och y' representerar 2D-representationen av en punkt i spelrymden, x och y är pixel-representationen samt w och h är skärmens bredd och höjd i pixlar. Division med 50 sker för att skala punkten till samma skala som övriga spelelement.

Efter detta skickades punkten vidare för att kollisiontestas mot samtliga objekt som är "Touchable". Om något objekt träffades sattes objektets position till samma som beröringens. Problemet med detta var att om fingret inte rörde på sig så skickades ingen ny information om att objektets position skulle vara oförändrad, utan endast då fingret rörde sig updaterades positionen på objektet. Utöver detta påverkades objektet fortfarande av accelerometern, så om fingret var stilla en stund så var sannolikheten stor att

objektet hade förflyttats utom räckhåll för beröringen. För att göra något åt detta sparades objektet som blev träffat i en variabel och punkten där beröringen befann sig i en annan variabel. På så vis sattes objektets position till den sparade punkten i varje frame. Om fingret avlägsnades så rensades variablerna och objektet fortsatte som vanligt. På så vis behövde fingret inte röra sig utan endast stanna på skärmen för att sätta objektets position. Ett antal problem kvarstod dock. Eftersom objektet fortfarande påverkades av accelerometern och alltså hade en kraft på sig så accelererade objektet konstant. Även om objektet förflyttades tillbaka till berörings-positionen i varje frame så rörde det sig, med accelererande hastighet, p.g.a. kraften från accelerometern. Utöver detta så kunde man få objekt att vara inuti varandra momentärt eftersom det var positionen på objektet som sattes. Objektet såg därför ut att studsa från fingerpositionen till en bit utanför objektet man höll den i. För att rätta till dessa problem så togs kraften skapad av accelerometern bort på det objektet som var påverkat av en beröring och istället för att sätta positionen på objektet applicerades en kraft från objektets mitt till fingret. Denna vektorn framtogs av vektorn från origo till objektet subtraherat med vektorn från origo till beröringen. Nu var implementationen av en beröring klar och kunde vidareutvecklas till multi-touch.

I multitouch-implementationen utvanns alla beröringar på skärmen och omvandlades var och en på samma sätt som för en beröring. Skillnaden var nu att en lista med punkter skickades vidare för att kollision-kontrolleras. Om kollision med någon/några av punkterna och objekten uppstod sparades objekten i en lista med fingrets koordinater som nyckel, för att hålla reda på vilken beröring som tillhör vilket objekt. Om beröringarna flyttade sig så skickade “Touches Moved” ut två listor med punkter, dels en lista med punkter som beröringarna hade varit i, och en lista med de nya punkterna. Listan med föregående punkter kontrollerades sedan mot listan med punkter och objekt ifall de hade något objekt registrerat till sig. I fallet då ett objekt var registrerat till punkten så avregistrerades den och registrerades istället till den nya positionen. Om inget objekt fanns registrerat så kollisions-testades punkten istället. Då någon beröring togs bort från skärmen skickade “Touches Ended” också ut två listor med föregående och nuvarande punkter. Båda listorna kontrollerades ifall något objekt var registrerat till dem och avregistrerades i så fall.

Den slutgiltiga implementationen klarar alltså av ett godtyckligt antal beröringar med modularitet för vad som händer med olika objekt.

5.6 Testning

Testning är alltid viktigt för att validera grundläggande och i slutändan anvancerad funktionalitet i en större kodbas. Arbetssättet i projektet har varit att kontinuerligt testa alla typer av förändringar man gör i koden,

oavsett om det är en bugg-fix, ny funktionalitet eller optimeringar.

5.6.1 Enhetstester

Inledningsvis skrevs omfattande enhetstester för att testa t.ex. vektor- och matrisbibliotek. Det gjordes för att mindre fel i dessa hade kunnat leda till väldigt stora och minst sagt svårhittade buggar längre fram i utvecklingsprocessen. Ett enhetstest innebär att man använder sig av funktioner och sedan m.h.a. påståenden (s.k. assertions) kontrollerar om utfallet från en funktion var korrekt. *STAssertTrue*, i figur 5.4 nedan, undersöker ifall funktionsanropet ger rätt resultat. Det booleska uttrycket *v2sub(vec1, vec2) == resultvec* ska i sådana fall vara *True*. Om fel värde returneras så kommer det uppstå ett felmeddelande i kompileringen av testkoden. Detta felmeddelande kommer då innehålla den text som ges i det andra argumentet till *STAssertTrue*.

```
STAssertTrue(v2sub(vec1, vec2) == resultvec, @"vec2sub inkorrekt");
```

Figur 5.4: Test för funktionen v2sub.

Att på detta vis hitta och täcka in alla potentiella "fallgroppar" var dock väldigt tidskrävande och därför implementerades även visuell testning.

5.6.2 Visuell testning

För att komplettera enhetstesterna med en enkel metod för att upptäcka saker som inte agerar som förväntat så implementerades stöd för visuell testning. Det första som gjordes för att säkerställa att detta fungerade korrekt var att rita ut pilar som representerade normaler och kontrollera att dessa var korrekta. Att testa visuellt istället för att skriva utförliga enhetstester ger inte 100% säkerhet att allt utförs korrekt och fungerar som det ska. Däremot är det en enkel metod för att upptäcka större fel, så som felvända normaler eller dylikt, istället för att sitta och försöka utläsa dessa fel direkt från flyttal. För oss var visuell testning en metod för att snabbt testa funktionalitet och hitta eventuella fel.

5.7 Optimering

Under projektets gång har det använts ett flertal olika metoder för att förbättra körtiden för koden i projektet. Framför allt har matriser varit fokus för många optimeringar då dessa används flitigt både i uträkningar för fysik och grafik. Alla förändringar har däremot inte varit fördelaktiga, utan tvärt om försämrat vissa delar i projektet på grund av felaktiga antaganden. Des-

sa felaktiga antaganden har efter hand rensats ut med hjälp av mätningar av programmets och vissa funktioners körtid.

5.7.1 Accelerate

Accelerate är ett ramverk som innehåller matematiska funktioner för digital signal-processering, matris- och vektorberäkningar, fft (fast fourier transform) och lösning av linjära ekvationssystem. Ramverket är optimerat för hårdvaran på iPhone, iPad och iPod Touch. Det var tänkt att ramverket skulle användas i vektor- och matrisbiblioteket i spelmotorn men efter prestandatester märktes det att de vanliga funktionerna skrivna i C ofta presterade ungefär lika bra som funktionerna i Accelerate. Detta beror på att Accelerate är optimerat för att behandla större mängder data på en och samma gång än vad som sker i spelmotorn, då det kan utföra beräkningar parallellt.

5.7.2 Optimeringar med assemblyspråk

Som nämnts under bakgrund så finns stora möjligheter till optimering med hjälp av SIMD-instruktioner på nyare enheter. Ett bibliotek med funktionerna har skrivits i assemblyspråk till dessa enheter. Dessa funktioner har matematiskt analyserats för parallella uträkningar och för att minska beroenden mellan instruktioner. Denna analys har gjorts för att vissa funktioner många gånger körs snabbare än motsvarande funktion skriven i C.

Kapitel 6

Resultat

Följande kapitel beskriver spelmotorns tillstånd vid projektets slut. Fokus för kapitlet ligger på de tekniska komponenterna i spelmotorn, dess slutgiltiga struktur och gränssnittet för att använda den. Beskrivning av prototyper som utvecklats för att visa upp spelmotorn ges också.

6.1 Strukturen i EvilEngine

Vår spelmotor består av 4 moduler med varsitt syfte. Dessa moduler är designade för att minimera databeroendet moduler emellan. Detta möjliggjordes genom att varje modul, i största möjliga utsträckning, endast modifierar sitt eget tillstånd och tar andra de modulernas data som argument till dess funktioner. Detta gör modulerna enkla att testa var för sig samt gör dem återanvändbara i framtida projekt.

Det som binder samman dessa moduler är EvilEngine, vilket består av en datatyp som innehåller alla modulers tillstånd, vilka objekt som för nuvarande är laddade och funktioner för att uppdatera detta huvudtillstånd. Motorn ansvarar även för minnesanvändningen, både för effektiv och förenklad användning. Detta görs med hjälp av en dynamisk lista, mer om dynamiska listor finns i sektion 4.1.1. Alla listor som hålls i spelmotorns tillstånd är av denna typ.

6.1.1 Moduler

De moduler som hittills har byggts är EvilRenderer, som ritar ut speltillståndet i OpenGL, EvilCollider, som simulerar kollisioner, EvilForces, som hanterar krafter, och EvilTimer, som räknar ut tiden mellan skärmuppdateringarna och utför planerade funktioner.

EvilRenderer

Denna modul använder sig av OpenGL ES version 1.1 eller 2.0 för att rita ut spelets nuvarande tillstånd. Vilken version som används för att rita ut bestäms av systemet och väljs vid uppstarten av programmet beroende på vilken version av OpenGL ES som enheten har stöd för. För närvarande stödjer modulen buffring och ljussättning av världen när version 2.0 används. Version 1.1 visar en enklare version för att vara kompatibel med äldre enheter.

Tillståndet innehåller en lista av alla objekt som ska ritas ut, med vilken version utav OpenGL ES samt hur nuvarande ljusmiljö ser ut. Datastrukturen som beskriver hur objekt ska ritas ut återfinns i sektion 6.1.2. Renderingskoden som används för OpenGL ES 2.0 finns inkluderad i bilaga B.

EvilCollider

Denna modul detekterar och löser kollisioner mellan objekt i spelvärlden. Kollisionsdetektionen är uppdelad i två faser, en approximativ och en exakt. I den approximativa fasen konstrueras cirklar som precis omsluter objekten. Cirklarna används sen för att approximera om två objekt kommer kollidera eller inte. Om en approximativ kollision detekteras körs också den exakta detektionen på objekten för att avgöra om de verkligen kolliderar och i så fall beräkna kontaktdata. Den exakta kollisionsdetektionen sker med hjälp av SAT (Separating Axis Theorem, detaljer i avsnitt 4.4.3). När alla kollisioner har detekterats så löses kontakterna en efter en av kollisionsberäknaren.

EvilForces

Denna modul applicerar krafter på objekt i spelvärlden. Krafter kan vara gravitation, konstanta krafter, fjäderkrafter som beror på avstånd, eller motståndskrafter som beror på vad för material som objektet passerar igenom.

Tillståndet innehåller en lista av krafter samt hur enheten är orienterad, och därmed också gravitationens riktning. Typen för krafter finns beskriven i sektion 6.1.2.

EvilTimer

Denna modul har en intern tid som används för att räkna ut tidssteg i simuleringen samt för att generera händelser vid planerade tidpunkter. Tillståndet innehåller en tidstämpel och en kö med planerade händelser.

6.1.2 Datatyper

Internt i spelmotorn används ett antal strukturer med olika funktion som är delar av speltillståndet. De viktigaste är RigidBody, som beskriver fysiska

egenskaper, SShape, som beskriver geometriska figurer, och DrawData, som beskriver hur objekt i spelvärlden ska renderas.

RigidBody

RigidBody innehåller de fysikaliska egenskaperna för ett objekt. Dessa används dels för att beräkna hur ett objekt ska integreras framåt, men också för att beräkna impulsen vid en kollision. Matrisen *bodyToWorldMatrix* (se fig 6.1) innehåller position, rotation och skalning för objektet och uppdateras vid varje integration. RigidBody har hand om matrisens minne, men delar en referens med bl.a. DrawData för att underlätta rendering.

```
typedef struct {
    float*  bodyToWorldMatrix;
    Vector2 velocity;
    Vector2 forceAccumulated;
    float   rotation;
    float   torqueAccumulated;
    float   inverseMomentOfInertia;
    float   inverseMass;
    float   damping;
    float   drag;
    float   restitution;
    float   friction;
} RigidBody;
```

Figur 6.1: Strukturen för RigidBody.

SShape

Detta är en samling polymorfiska strukturer som delar några fält, och vars typ testas för att ta reda på vilka egenskaper och andra fält som just denna instans av strukturen har. I spelmotorn finns två undertyper till SShape, en som beskriver cirklar och en annan som beskriver polygoner. Alla dessa skickas som pekare till SShape men kastas sedan till undertyper när deras egenskaper behöver läsas. Här nedan i figur 6.2 ges definitionen för polygonstrukturen. Denna typ används av EvilCollider för att detektera kollisioner.

```
typedef struct
{
    const enum ShapeType type;
    float*               transShapeToBody;
    const float          radius;
    const int            pointSize;
    const float*         hullEdges;
    const int            hullLength;
} SPolygon;
```

Figur 6.2: Strukturen för SPolygon.

DrawData

Även DrawData är en samling polymorfiska strukturer som delar alla fält. Fälten används olika beroende på vad för typ objektet är. Strukturen beskriver dels hur ett objekt med färg eller textur ska renderas. Den bestämmer också om objektet ska sparas i grafikminnet, renderas med ljusmiljö eller både och. För att dessa ska vara länkade till objekt i spelvärlden så innehåller alla dessa pekare till två matriser, en för transformationen av RigidBody och en för en transformationen av SShapes. Detta skapar ett databeroende till både RigidBody och SShape, men kan undvikas genom att peka till fristående matriser istället, vilket används för att rendera objekt som inte ska påverkas av varken krafter eller kollisioner. Dessa olika typer bestäms utav en enum som visas i figur 6.3 här nedan.

```
typedef enum DrawType {
    NODRAW = 0,
    COLOURED,
    COLOUREDBUFF,
    COLOUREDLIGHT,
    COLOUREDLIGHTBUFF,
    TEXTURED,
    TEXTUREDBUFF,
    TEXTUREDLIGHT,
    TEXTUREDLIGHTBUFF,
} DrawType;
```

Figur 6.3: DrawType enum.

ForceGenerator

ForceGenerator är en samling polymorfiska strukturer som beskriver olika krafter och hur de ska appliceras i spelvärlden.

EvilBody

EvilBody är en struktur som återfinns i huvudtillståndet i EvilEngine och beskriver ett objekt i spelvärlden. Denna struktur är en av de få där databeroende mellan moduler måste finnas. Till varje EvilBody hör en RigidBody, en lista med SShape-strukturer, listor med DrawData- och ForceGenerator-referenser, samt pekare till funktioner som anropas vid händelser. Detta illustreras i kodexempel 6.4 nedan.

```
typedef struct EvilBody {
    RigidBody       body;
    BodyUpdateFunc  updateFunc;
    BodyCollideFunc collideFunc;
    BodyTouchFunc   touchFunc;
    DynamicArray    shapes;
    DynamicArray    forceIndices;
    DynamicArray    drawIndices;
} EvilBody;
```

Figur 6.4: EvilBody datastruktur.

6.1.3 API

Spelmotorn styrs med hjälp av ett mycket enkelt API vars huvudsyfte har blivit att snabbt möjliggöra nya funktioner. APIn anropas med funktioner som tar nuvarande tillstånd som första argument, en typsäker referens (sektion 6.1.3) till det objekt som ska påverkas och eventuellt andra argument. Nedan i figur 6.5 finns exempel på några funktioner. De tre första är till för att registrera samt associera krafter, medan de två sista är till för att flytta objekt med transformationsmatriser.

Med nuvarande API kan man lägga till nya objekt i världen, associera geometriska figurer med dem, bestämma hur objekten ska ritas ut, lägga till gravitation till objekten, manuellt flytta objekt i världen, samt lägga till händelsefunktioner.

```
ForceRef addEvilForce(EvilGameState* state, ForceGenerator* generator);

int addEvilForceOnBody(EvilGameState* state,
                       BodyRef  bodyRef,
                       ForceRef forceRef);

int addEvilGravity(EvilGameState* state, BodyRef bodyRef);

int applyMatrixToBody(EvilGameState* state,
                      BodyRef bodyRef,
                      float matrix[16]);

int applyMatrixToShape(EvilGameState* state,
                       ShapeRef shapeRef,
                       float matrix[16]);
```

Figur 6.5: Exempel på funktioner ur EvilEngines API.

Typsäkra referenser

Då all minneshantering sköts av spelmotorn internt med listor så refereras alla objekt med index i dessa. Dessa index är alla av samma typ, men omsluts av en struktur utanför motorn, vilket här kallas för en typsäker referens. Det finns fyra av dessa typer, BodyRef, ShapeRef, ForceRef och DrawRef. Funktioner i API tar endast en av dessa var, och det finns ingen risk att de förväxlas då kompilatorn varnar när fel referens används.

6.2 Vektor- och matrisbibliotek

En av möjligheterna med plattformen som observerades i sektion 2.2 var att nyare enheter med ARMv7-baserade processorer hade hårdvarustöd för operationer på vektorer parallellt med så kallade SIMD-instruktioner. Detta har tagits tillvara på i ett optimerat bibliotek för vektor- och matrisoperationer där vissa funktioner har skrivits endast i assemblyspråk och vissa med assemblyspråk inbäddat i C-filer. I figur 6.6 nedan ges implementationen av vektorer multiplicerat med matriser. Implementationen av matrismultiplikation återfinns i bilaga A. Dessa funktioner har även en C-variant för kompatibilitet med äldre enheter.

De funktioner som är optimerade är matrismultiplikation samt multiplikation mellan vektorer och matriser. Under optimala förhållanden då matriserna ej ändras mellan anropen kan upp till sexton optimerade funktioner utföras under samma tid som en funktion i ren C.

```
vldmia %[vmem], { q1 }
vldmia %[mmem], { q8-q11 }
vmul.f32 d0, d16, d2[0]
vmla.f32 d0, d18, d2[1]
vmla.f32 d0, d20, d3[0]
vmla.f32 d0, d22, d3[1]
```

Figur 6.6: Assemblykod för multiplikation av en matris och en vektor.

6.3 Prototyper

För att visa projektets fortgång har vi skapat två prototyper som visar den funktionalitet som har implementerats i spelmotorn.

6.3.1 Prototyp 1

Till halvtidsredovisningen utvecklades prototyp 1. Syftet med denna var att visa upp kollisionshantering, enkel rendering av former och interaktion med hjälp utav accelerometern. Prototypen innehöll enkla geometriska figurer som kunde kollidera med varandra och som påverkades av gravitationen. Gravitationen i sin tur styrdes av accelerometern, vilket gjorde att figurernas hastigheter påverkades när telefonen lutades eller rörde på sig. En lista över funktionalitet som demonstrerades i prototyp 1 återfinns nedan.

- Gravitation
- Enkla geometriska figurer
- Kollisionshantering
- Inelastiska kollisioner
- Stöd för accelerometern
- OpenGL-rendering

En bild från den första prototypen återfinns i figur 5.1 under sektion 5.3.3.

6.3.2 Prototyp 2

Inför slutredovisningen utvecklades en andra prototyp för att demonstrera funktionalitet som har lagts till sedan prototyp 1. Prototyp 2 innehåller, förutom utökad funktionalitet, även förbättringar och eliminering av buggar i den funktionalitet som redan fanns implementerad i prototyp 1. Prototyp 2 består av en demonstrationsbana med ett flertal polygoner och fyrkanter. Ovanpå varje figur finns en textur, som ger figurerna en snyggare och mer verklighetstrogen yta. Figurerna styrs likt prototyp 1 av accelerometern och

kolliderar med varandra. På demonstrationen har det dessutom lagts till en bakgrundsbild som syns i bakgrunden och väggar som visar var kanterna av banan finns. Den funktionalitet som är implementerad i prototyp 2 återfinns i nedanstående lista.

- Gravitation
- Konvexa polygoner, väggar och bakgrund
- Kollisionshantering (förbättrad)
- Inelastiska kollisioner
- Stöd för accelerometer
- OpenGL-rendering
- Texturer
- Ljuseffekter

I figur 6.7 visas prototyp 2 och stora delar av den funktionalitet som har implementerats.

Figur 6.7: Bild på prototyp 2

Kapitel 7

Diskussion

Målet med projektet var att designa och implementera en fullständig spelmotor som kan användas för att bygga spel till framförallt iPhone men också andra mobila plattformar. Spelmotorn är så gott som fullständig men behöver mer arbete. I detta kapitel diskuteras val och prioriteringar som gjordes under projektets gång, vad vi kunde gjort annorlunda och hur det i slutändan hade påverkat spelmotorn.

7.1 Planering och arbetsprocess

Som tidigare nämnts i avsnitt 3.1 så har utvecklingen varit uppdelad i två större iterationer och flera mindre där först grundläggande fysik och grafik skulle implementeras och sedermera utvecklas till att vara betydligt mer avancerad samtidigt som ljud och interaktion skulle inkluderas. Denna modell fungerade bra till en början men när vi började experimentera med alternativa metoder och fick skriva om stora kodavsnitt märkte vi att vi inte hade planerat för att skriva kod som senare inte skulle användas. Detta har medfört att de områden som vi tyckte var mindre viktiga inte har fått så mycket fokus som de borde för att senare kunna användas direkt för att utveckla ett spel.

Från början var tanken att ett större demo med flertalet banor skulle konstrueras för att visa upp spelmotorn. En konceptdesign skulle arbetas fram som kunde ligga som grund för banorna och karaktärerna i demot. Detta är också en del av projektet som kontinuerligt har förlorat prioritet under utvecklingens gång när gruppens intresse för att prova nya tekniker och optimera spelmotorns struktur och funktionalitet har tagit över. Av denna anledning blev prototyperna istället en väldigt enkel uppvisning av vad spelmotorn är kapabel till, utan förfinad design och flertalet banor. Det som har gjorts i form av koncept och vad som senare ska utvecklas till en spelidé beskrivs under avsnitt 8.1.

Intresset för att prova olika tekniker och optimeringar har inte bara

medfört att vissa delar i projektet har förlorat prioritet. Det har också gjort att gruppens kunskap i de områden där utvecklingen främst har pågått ständigt har vuxit. Gruppmedlemmarna har också fått bred erfarenhet av de programmeringsspråk som har använts, framför allt C och Objective-C men också GLSL (OpenGL Shading Language). Erfarenheterna har lett till större förståelse för de konventioner som finns och för optimeringar som är möjliga språkmässigt.

7.2 Val av teknik

7.2.1 Objective-C och C

Portabilitet och plattformsoberoende var viktiga begrepp redan från början av projektet. Det som inte insågs då var hur mycket av slutresultatet som faktiskt skulle vara just plattformsoberoende. Kodbasen har gått från att nästan uteslutande vara skriven i Objective-C till att innehålla väldigt lite och istället huvudsakligen bestå av ren C-kod. Övergången har inte varit helt problemfri och har krävt stora omstruktureringar men samtidigt inneburit märkbara prestandaförbättringar. Man kan diskutera varför vi inte valde t.ex. C++ istället då det fungerar på ungefär samma plattformar som C och har liknande prestanda. En anledning är att det inte finns speciellt många kompletta spelmotorer skrivna i C vilket gör att vår spelmotor utmärker sig i mängden.

7.2.2 OpenGL

Beslutet att använda OpenGL baserade sig från början i möjligheten att rendera djup, trots att utvecklingen i övrigt fokuserade på grafikrendering i 2 dimensioner. Ett av alternativen till OpenGL var *Quartz 2D* som är en del av ramverket *Core Graphics* och finns tillgängligt på Mac OS X och iOS. Quartz 2D är ett lager ovanpå OpenGL, specialiserat för att rendera 2D-grafik. Det insågs senare att OpenGL var ett bra val även om djup inte blev så viktigt i slutändan. I OpenGL finns bland annat större möjligheter att skräddarsy grafiska effekter såsom ljus och skuggor med hjälp av shaders. Att använda OpenGL ger även större plattformsoberoende, eftersom OpenGL till skillnad från Quartz 2D finns tillgängligt på andra plattformar än Apples.

7.2.3 Externa bibliotek

Som en följd av att gruppen ville lära sig så mycket som möjligt och experimentera med olika tekniker har användandet av färdiga bibliotek varit mycket begränsat. Det som har använts är standardbiblioteken i C och interna ramverk i iOS som *Foundation*, *CoreFoundation*, *UIKit* och självfallet *OpenGL ES*. Foundation och CoreFoundation används i den filhantering

och filmanipulering som behövs för att läsa in texturer och shaders. UIKit används bara för att presentera den vy där renderingen sker.

7.3 Ljudmotor

Målet med projektet var, som tidigare har nämnts, att ha en komplett spelmotor som var redo för att bygga ett spel på. Att bygga ett spel kräver i princip att det finns en ljudmotor, för utan ljud blir spelet genast relativt tråkigt. Trots målet att ha med ljudmotor så har vi valt att prioritera bort detta. Anledningen till det är främst att det fanns annat viktig funktionalitet att implementera, samt att ljudmotorn inte kan göras plattformsoberoende. Det har hela tiden varit ett mål att hålla majoriteten av koden abstraherbar för att motorn ska kunna användas på andra plattformar än iOS. Orsaken till att det inte går att skriva en ljudmotor som är plattformsoberoende i C är att det kräver att man kontaktar operativsystemet och använder detta för att spela ljud. Eftersom vi utvecklar mot iOS så kommer detta vara beroende av mjukvaran som är tillgänglig där och detta kommer inte anropas på samma sätt i t.ex. Android eller andra operativsystem. Vi har därför valt att lägga ljudmotorn på is och placera den i framtida arbete.

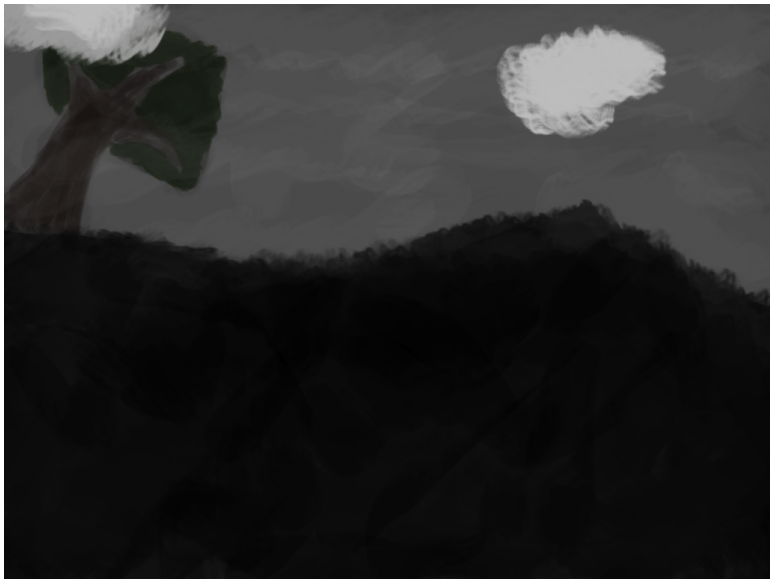
Kapitel 8

Framtida arbete

I detta kapitel beskrivs hur vi har tänkt att fortsätta utvecklingen av spelmotorn efter att projektet är avslutat.

8.1 Spelidén

Innan vi beslutade att göra en spelmotor istället för ett spel hann vi diskutera lite idéer till ett eventuellt spelprojekt. Spelen som nämns i 2.3 har till stor del influerat även våra tankar gällande spelkoncept. Om det framöver blir aktuellt att fortsätta arbetet i gruppen så finns det en god chans att vi kommer utveckla ett spel. Spelkonceptet i stora drag är att det skulle bli ett äventyrsspel med dystra miljöer liknande *Limbo* [13]. Uppdragen ska gå ut på att ta sig från punkt A till B och lösa diverse pusselproblem på vägen. Gravitationsskiften ska vara ett centralt verktyg för att lösa pusslen, spel som influerat detta finns i sektion 2.3.3. Några skisser på koncept till miljöer i spelet finns nedan i figur 8.1 och 8.2. Dessa gjordes innan projektets riktning ändrades till att skapa en spelmotor istället för ett spel, men fanns kvar som inspiration till ett mer avancerat demo medan det fortfarande var prioriterat.



Figur 8.1: Dyster bakgrundsmiljö

Figur 8.2: Påbörjad stadssiluett

8.2 Ljudmotor

Vi kommer i framtiden även utveckla en ljudmotor, troligtvis i Objective-C, som kan användas tillsammans med resterande spelmotor på iOS-plattformen. Detta för att en ljudmotor kommer behövas om vi ska bygga ett spel till iP-

hone. Den kommer troligtvis hållas relativt simpel, men ha lite realistiska effekter som positionerade ljudkällor som gör att volymen ökar och avtar beroende på om man rör sig mot eller från ljudkällan. M.h.a. detta skulle man tex kunna simulera en svärm med flugor eller en radio.

8.3 Distribution

Vi har inom gruppen diskuterat om vi på något sätt ska distribuera spelmotorn när den är färdig. Ingen slutsats har nåtts men diskussioner kring att släppa det som öppen källkod med t.ex. *GPL-licens* [21] som enkelt uttryckt är ett sätt att sprida öppen källkod på ett sådant sätt att den som använder sig av koden aldrig kan distribuera sin kod som sluten källa utan denna ärver licensen och måste därmed dela sin kod som öppen källkod också. Vi tror detta skulle bidra till användandet av motorn eftersom användaren får modifiera motorn precis hur som helst för att passa de behov som stöts på under utvecklingsprocessen. Vi tror även att det kan vara en mycket god referens för framtida syften att kunna visa upp ett omfattande kodprojekt som förhoppningsvis andra använder sig av.

8.4 Scriptning av banor

Ifall motorn ska användas för att skriva ett spel med många banor, kommersialiseras eller distribueras som öppen källkod så kan det vara bra att skriva någon typ av scriptspråk för att förenkla processen för att bygga upp banor. Våran plan är att skriva ett scriptspråk som använder sig av keywords, såsom “new” och “destroy” m.fl., för att lägga till och ta bort objekt i spelvärlden. Detta är särskilt viktigt om den ska användas av andra än oss, för att dessa ska slippa sätta sig in i hela motorn för att veta vilka funktioner som ska användas för en så simpel sak som att lägga till en rektangel med fysikaliska engenskaper i mitten av skärmen. Det skulle med scriptspråk kunna se ut på följande sätt: *new rectangle side1 side2 position physicsOn;*. Dessa script-kommandon skulle sedan parsas till C-kod som använder sig av funktionerna i vår motor för att skapa detta objekt och placera det rätt. Denna kod skulle vara på minst 20-30 rader kod i C för anrop till uppbyggnad av hörn, triangulisering och applicering av krafter o.s.v.

8.5 Kontinuerlig kollisionsdetektion

I spelmotorns nuvarande tillstånd används diskret kollisionsdetektion. Det innebär att kollisioner bara detekteras vid de diskreta tidsteg som definieras av uppdateringsfrekvensen. Diskret kollisionsdetektion medför också att kollisioner inte detekteras förrän de inblandade objekten redan överlappar varandra. Med kontinuerlig kollisionsdetektion försöker fysikmotorn förutspå

kollisionerna så att de kan lösas så tidigt som möjligt [23], lämpligen innan objekten överlappar varandra för att undvika de problem som överlappning innebär (se avsnitt 5.3.9). Det finns ett par olika metoder som används för kontinuerlig kollisionsdetektion:

- *Gilbert–Johnson–Keerthi distance algorithm (GJK)*
  Beräknar avståndet mellan två konvexa polygoner. Avståndet kan sedan användas för att approximera när kollisionen ska ske [25].

- *Ray Casting*
  Genom att skjuta en stråle (linje) på två objekts Minkowski-summa i samma riktning som deras relativa hastighet kan man approximera exakt när och var de två objekten kommer kollidera [26].

Metoderna har sina för och nackdelar och vi behöver undersöka dem noggrannare innan vi vet vilken som passar bäst för vår spelmotor.

Kontinuerlig kollisionsdetektion är en förutsättning för att kunna använda spekulativa kontakter. *Iterative Dynamics with Temporal Coherence* [20] beskriver bland annat hur spekulativa kontakter och tidskoherens kan användas för att implementera ett stabilt kollisionssystem. När spekulativa kontakter används löses en kollision uppdateringen innan den sker. En impuls appliceras för att bromsa objekten så att de precis vidrör varandra vid nästa uppdatering. Detta minskar förekommandet av mikrokollisioner (avsnitt 5.3.9) och gör kollisionssystemet betydligt stabilare [22].

8.6 Ytterligare optimering

Vi har under projektets gång försökt att tänka på prestandakrav i olika implementationer och väga dessa mot varandra. Dock har optimeringen inte fått gå ut över framsteg allt för mycket. Det innebär att det finns en hel del kvar att göra på denna fronten, som tidigare nämnts i sektion 5.7 har en del kod skrivits om till assemblyspråk med signifikant kostnadsminskning, det finns dock fler delar som skulle kunna effektiviseras m.h.a. assembly.

Skuggor är en annan funktion där optimering ännu inte har hunnits med överhuvudtaget. Det krävs en hel del mer research innan vi vet vilket spår vi ska följa för att få detta att fungera bra. Dock har diskussioner förts kring *shadowmaps* som är texturer som beräknas för varje bilduppdatering. Dessa texturer är relativt lågupplösta för att hålla prestanda-kostnaden nere och innehåller information om var skuggor ska renderas. Detta kommer antagligen kunna lösas på liknande sätt som med ljusets *lightmaps*.

8.7 Vatten och rök

I framtida arbete kommer vi fortsätta med implementation av rök och vatten genom fluiddynamik. Det som främst krävs är utökning av algoritmerna för

att de även ska fungera för vatten, samt att optimera kod och algoritmer för att uppnå en högre uppdateringsfrekvens när fluiden renderas.

Kapitel 9

Slutsats

Utvecklingen av en spelmotor är en avancerad och tidskrävande process. Många olika matematik-, fysik- och programmeringsrelaterade ämnen måste studeras och evalueras innan tillämpning för att få ett bra resultat. Om en noggrann utredning ej sker innan någonting läggs till eller ändras är risken stor att det måste skrivas om eller ändras igen vid ett senare tillfälle för att få önskade egenskaper. Spelmotorer kan se ut på väldigt många olika sätt beroende på vad de ska utföra. Val av teknik och implementationer av olika delar i en spelmotor är inte alltid så enkelt och om man väljer att gå en egen väg är implementering och optimering ännu svårare än om man väljer en redan existerande och beprövad metod.

Även om EvilEngine i nuläget inte är redo för att användas direkt för att utveckla ett spel så har arbetet i projektet resulterat i en mycket stabil grund. Spelmotorn är strukturerad på sådant sätt att den mycket lätt kan utökas samtidigt som gruppen har erhållit en bred kunskapsbas för vidare utveckling av spelmotorn och spelutveckling generellt.

Arbete i grupp kan vara både fördelaktigt då uppdelning av arbetet kan ske för att avlasta varandra, men vissa nackdelar finns även också. En nackdel är att man måste hitta något bra sätt att hålla alla gruppmedlemmar uppdaterade om vad som har utförts eller ändrats. Det kan vara en tidskrävande process i sig, men är en viktig del i utförandet om ett samarbete inom gruppen ska fungera.

Bilaga A

Matrismultiplikation i assemblyspråk

```
#ifdef __ARM_NEON__
        @ Setup file code type and alignment.
        .thumb
        .align  2

        .globl          _load_m0_neon
        .private_extern _load_m0_neon
        .thumb_func     _load_m0_neon
        @ Load m0 matrix into NEON registers.
        @ { q12-q15 } = m0
_load_m0_neon:
        vldmia  r0, { q0 - q3 }
        mov pc, lr

        .globl          _load_m1_neon
        .private_extern _load_m1_neon
        .thumb_func     _load_m1_neon
        @ Load m1 matrix into NEON registers.
        @ { q18-q11 } = m1
_load_m1_neon:
        vldmia  r0, { q8 - q11 }
        mov pc, lr

        .globl          _load_d0_neon
        .private_extern _load_d0_neon
        .thumb_func     _load_d0_neon
        @ Load d0 matrix into NEON registers.
        @ { q0 - q3 } = d0
_load_d0_neon:
        vldmia  r0, { q12 - q15 }
        mov pc, lr

        .globl          _store_m0_neon
        .private_extern _store_m0_neon
        .thumb_func     _store_m0_neon
```

```
        @ Store matrix from NEON registers.
_store_m0_neon:
        vstmia  r0, { q0 - q3 }
        mov     pc, lr

        .globl          _store_m1_neon
        .private_extern _store_m1_neon
        .thumb_func     _store_m1_neon
        @ Store matrix from NEON registers.
_store_m1_neon:
        vstmia  r0, { q8 - q11 }
        mov     pc, lr

        .globl          _store_d0_neon
        .private_extern _store_d0_neon
        .thumb_func     _store_d0_neon
        @ Store matrix from NEON registers.
_store_d0_neon:
        vstmia  r0, { q12 - q15 }
        mov     pc, lr

        .globl          _move_d0_m0_neon
        .private_extern _move_d0_m0_neon
        .thumb_func     _move_d0_m0_neon
        @ Move d0 data into m0 registers.
_move_d0_m0_neon:
        vmov    q0, q12
        vmov q1, q13
        vmov q2, q14
        vmov q3, q15
        mov pc,  lr

        .globl          _move_d0_m1_neon
        .private_extern _move_d0_m1_neon
        .thumb_func     _move_d0_m1_neon
        @ Move d0 data into m1 registers.
_move_d0_m1_neon:
        vmov q8,  q12
        vmov q9,  q13
        vmov q10, q14
        vmov q11, q15
        mov pc,  lr

        .globl          _transpose_m0_neon
        .private_extern _transpose_m0_neon
        .thumb_func     _transpose_m0_neon
        @ Transpose m0 in { q0 - q3 } registers.
        @ m0 = m0^T
_transpose_m0_neon:
        @ Transpose internal matrices.
        vtrn.32 q0, q1
        vtrn.32 q2, q3
        @ Swap snd and thrd internal matrices.
        vswp    d1, d4
```

```
        vswp    d3, d6
        mov pc, lr

        .globl          _multm4m4_neon
        .private_extern _multm4m4_neon
        .thumb_func     _multm4m4_neon
        @ Matrix multiplication
        @ m0 in {q0-q3},  column major
        @ m1 in {q8-q11}, column major
        @ Resulting matrix in d0 {q12-q15}
        @ d0 = m0 * m1
        @ Rows of m0 is multiplied by columns in m1.
_multm4m4_neon:
        @ Multiply fst row of m0 with fst column of m1.
        vmul.f32    q12, q8, d0[0]
        vmul.f32    q13, q8, d2[0]
        vmul.f32    q14, q8, d4[0]
        vmul.f32    q15, q8, d6[0]
        @ Multiply snd row of m0 with snd column of m1.
        vmla.f32    q12, q9, d0[1]
        vmla.f32    q13, q9, d2[1]
        vmla.f32    q14, q9, d4[1]
        vmla.f32    q15, q9, d6[1]
        @ Multiply trd row of m0 with trd column of m1.
        vmla.f32    q12, q10, d1[0]
        vmla.f32    q13, q10, d3[0]
        vmla.f32    q14, q10, d5[0]
        vmla.f32    q15, q10, d7[0]
        @ Multiply fth row of m0 with fth column of m1.
        vmla.f32    q12, q11, d1[1]
        vmla.f32    q13, q11, d3[1]
        vmla.f32    q14, q11, d5[1]
        vmla.f32    q15, q11, d7[1]
        mov pc, lr
#endif
```

Bilaga B

Renderare för OpenGL ES 2.0

```
#include <OpenGLES/ES2/gl.h>
#include <OpenGLES/ES2/glext.h>

#include "RendererCommon.h"
#include "MatrixOps.h"

/* Load lightMap texture.
 */
void loadLightMap(RenderData* renderer, DrawData* draw);

/* Load uniforms for coloured draw data.
 */
void loadUniformsColoured(RenderData* renderer, DrawColoured* colour);

/* Setup vertex and colour arrays.
 */
void enableColouredData(RenderData* renderer, DrawColoured* colour);

/* Load uniforms for textured draw data.
 */
void loadUniformsTextured(RenderData* renderer, DrawTextured* textured);

/* Setup vertex and texture coordinate arrays.
 */
void enableTexturedData(RenderData* renderer, DrawTextured* textured);

void renderES2(RenderData* renderer, DrawData* draw) {
    switch (draw->type) {
    case COLOUREDLIGHT:
        loadLightMap(renderer, draw);
    case COLOURED:
        loadUniformsColoured(renderer, (DrawColoured*)draw);
        enableColouredData(renderer, (DrawColoured*)draw);
        renderPrimitives(draw);
        break;
```

```
    case COLOUREDLIGHTBUFF:
        loadLightMap(renderer, draw);
    case COLOUREDBUFF:
        loadUniformsColoured(renderer, (DrawColoured*)draw);
        if (enableBuffer(draw) == 0)
            enableColouredData(renderer, (DrawColoured*)draw);
        renderPrimitives(draw);
        disableBuffer();
        break;
    case TEXTUREDLIGHT:
        loadLightMap(renderer, draw);
    case TEXTURED:
        loadUniformsTextured(renderer, (DrawTextured*)draw);
        enableTexturedData(renderer, (DrawTextured*)draw);
        renderPrimitives(draw);
        break;
    case TEXTUREDLIGHTBUFF:
        loadLightMap(renderer, draw);
    case TEXTUREDBUFF:
        loadUniformsTextured(renderer, (DrawTextured*)draw);
        if (! enableBuffer(draw))
            enableTexturedData(renderer, (DrawTextured*)draw);
        renderPrimitives(draw);
        disableBuffer();
        break;
    default:
        break;
    }
}
void loadLightMap(RenderData* renderer, DrawData* draw)
{
    ShaderContainer* shader    = loadShader(&renderer->shaderMng, draw->program);
    GLuint *uniforms           = shader->uniforms;
    const GLuint lightMap      = draw->lightMap;
    const GLfloat* lightMatrix = draw->lightMatrix;

    glUniformMatrix4fv(uniforms[UNI_LIGHTMAP_MAT], 1, GL_FALSE, lightMatrix);

    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, lightMap);
    glUniform1i(uniforms[UNI_LIGHTMAP_SAMPLER], 1);
}

void loadUniformsColoured(RenderData* renderer, DrawColoured* colour)
{
    const float* shapeToBody = colour->transShapeToBody;
    const float* bodyToWorld = colour->transBodyToWorld;

    ShaderContainer* shader = loadShader(&renderer->shaderMng, colour->program);
    GLuint *uniforms = shader->uniforms;

    float modelWorldMatrix[16];
    multm4x4(modelWorldMatrix, shapeToBody, bodyToWorld);
```

```
    glUniformMatrix4fv(uniforms[UNI_MODELWORLD_MAT], 1, GL_FALSE, modelWorldMatrix);
}

void enableColouredData(RenderData* renderer, DrawColoured* colour)
{
    HashMap* resourceMap       = &renderer->resourceMap;
    const GLsizei  length      = colour->length;
    const GLfloat* vertices    = colour->vertices;
    const GLint    vsize       = colour->vsize;
    const GLubyte* colours     = colour->colours;
    const GLint    csize       = colour->csize;

    if (isBufferedDraw((DrawData*)colour)) {
        bindFloatBufferObject(resourceMap, vertices, vsize, length);
        vertices = NULL;
    }
    glVertexAttribPointer(ATTRIB_VERTEX, vsize, GL_FLOAT, 0, 0, vertices);
    glEnableVertexAttribArray(ATTRIB_VERTEX);

    if (isBufferedDraw((DrawData*)colour)) {
        bindUByteBufferObject(resourceMap, colours, csize, length);
        colours = NULL;
    }
    glVertexAttribPointer(ATTRIB_COLOR,
                          csize,
                          GL_UNSIGNED_BYTE,
                          GL_TRUE,
                          0,
                          colours);
    glEnableVertexAttribArray(ATTRIB_COLOR);
}

void loadUniformsTextured(RenderData* renderer, DrawTextured* textured)
{
    const float*   shapeToBody = textured->transShapeToBody;
    const float*   bodyToWorld = textured->transBodyToWorld;
    const float*   coordMatrix = textured->transCoordinates;
    const GLuint   texture     = textured->texture;

    ShaderContainer* shader = loadShader(&renderer->shaderMng, textured->program);

    GLuint *uniforms = shader->uniforms;

    float modelWorldMatrix[16];
    multm4x4(modelWorldMatrix, shapeToBody, bodyToWorld);

    glUniformMatrix4fv(uniforms[UNI_MODELWORLD_MAT], 1, GL_FALSE, modelWorldMatrix);
    glUniformMatrix4fv(uniforms[UNI_TEXTURE_MAT], 1, GL_FALSE, coordMatrix);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    glUniform1i(uniforms[UNI_TEXTURE_SAMPLER], 0);
}
```

```
void enableTexturedData(RenderData* renderer, DrawTextured* textured)
{
    HashMap* resourceMap    = &renderer->resourceMap;
    const GLsizei  length   = textured->length;
    const GLfloat* vertices = textured->vertices;
    const GLint    vsize    = textured->vsize;
    const GLubyte* coords   = textured->coordinates;
    const GLint    csize    = textured->csize;

    if (isBufferedDraw((DrawData*)textured)) {
        bindFloatBufferObject(resourceMap, vertices, vsize, length);
        vertices = NULL;
    }
    glVertexAttribPointer(ATTRIB_VERTEX,
                          vsize,
                          GL_FLOAT,
                          GL_FALSE,
                          0,
                          vertices);
    glEnableVertexAttribArray(ATTRIB_VERTEX);

    if (isBufferedDraw((DrawData*)textured)) {
        bindUByteBufferObject(resourceMap, coords, csize, length);
        coords = NULL;
    }
    glVertexAttribPointer(ATTRIB_TEXCOORD,
                          csize,
                          GL_UNSIGNED_BYTE,
                          GL_TRUE,
                          0,
                          coords);
    glEnableVertexAttribArray(ATTRIB_TEXCOORD);
}
```

Litteraturförteckning

[1] Unreal Engine
`http://udk.com/` (2011-05-09)

[2] Dokumentation ARM11 (2011)
`http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/`
(2011-04-06)

[3] Dokumentation OpenGL ES1 (2011)
`http://www.khronos.org/registry/gles/` (2011-04-06)

[4] Dokumentation Cortex-A8 (2011)
`http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/`
(2011-04-06)

[5] Dokumentation OpenGL ES2 (2011)
`http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/`
(2011-04-06)

[6] Millington, I. (2007) *Game physics engine development*, San Francisco, Elsevier.

[7] L. Wei-Meng (2010) *Beginning iPhone SDK: Programming with Objective-C*, Indianapolis, Wiley Publishing.

[8] Pagh, R. Rodler, F. (2004) *Cuckoo hashing*, Journal of Algorithms 51, pp. 122–144.

[9] Hecker, C. (1996) *Physics, Part 1: The Next Frontier*, Game Developer Magazine, Dec/Jan 1996, pp. 12-20.

[10] Hecker, C. (1996) *Physics, Part 2: Angular Effects*, Game Developer Magazine, Oct/Nov 1996, pp. 14-22.

[11] Hecker, C. (1997) *Physics, Part 3: Collision Response*, Game Developer Magazine, Feb/Mar 1997, pp. 11-18.

[12] Neumann, E. (2004) *My Physics Lab*
`http://www.myphysicslab.com/` (2011-04-07)

[13] Playdead (2011)
`http://limbogame.org` (2011-04-07)

[14] Sparr, G. (1994) *Linjär Algebra*, Lund, Studentlitteratur.

[15] Bilting, U. Skansholm, J. (2000) *Vägen till C*, Lund, Studentlitteratur.

[16] Antonio, F. (1992) *Faster line segment intersection*, I: Kirk, D. *Graphic Gems 3*, pp. 199-202. Orlando, USA, Academic Press.

[17] Box2D (2007)
`http://www.box2d.org/` (2011-05-09)

[18] Stam, Jos. (2003) *Real-Time Fluid Dynamics for Games*, Game Developer Conference 2003, San Jose, CA, USA 6-8 September

[19] Juul, J. (2009) *A Casual Revolution: Reinventing Video Games and Their Players*, The MIT Press, Cambridge, MA, USA.

[20] Catto, E (2005) *Iterative Dynamics with Temporal Coherence*
`http://www.bulletphysics.com/ftp/pub/test/physics/papers/IterativeDynamics.pdf` (2011-05-09)

[21] GNU GPL-licens (2011)
`http://www.gnu.org/licenses/quick-guide-gplv3.html` (2011-05-09)

[22] Firth, P. (2011) *Speculative Contacts – a continuous collision engine approach*
`http://www.wildbunny.co.uk/blog/2011/03/25/speculative-contacts-an-continuous-collision-engine-approach-part-1/` (2011-05-16)

[23] Firth, P. (2011) *Collision detection for dummies*
`http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/` (2011-05-16)

[24] Eberly, D. (2001) *Intersection of Convex Objects: The Method of Separating Axes*
`http://www.geometrictools.com/`

[25] Van Den Bergen, G. (1999) *A Fast and Robust GJK Implementation for Collision Detection of Convex Objects*, Department of Mathematics and Computing Science, Eindhoven University of Technology

[26] Van Den Bergen, G. (2004) *Ray Casting against General Convex Objects with Application to Continuous Collision Detection*, Playlogic Game Factory, Breda, Netherlands

[27] Dimitrov, R. (2007) *Cascaded Shadow Maps*, Nvidia Corporation, Santa Clara, CA, USA

Ordlista

advektion Beskriver hur ett ämne transporteras i en fluid som exempelvis sotpartiklar eller sand.

binär heap Partiellt ordnat vänsterbalanserat träd, detta innebär att det är fullt i alla nivåer utom eventuellt den djupaste nivån som i sådant fall är fylld från vänster.

bounding volume En volym, ofta formad som någon vanlig geometrisk figur (t.ex. cirkel eller rektangel), som helt och hållet omsluter en godtyckligt formad kropp. Används för att jämförelser om övelappning ofta är enkla på geometriska figurer.

C Generellt, imperativt programspråk. C är ett av de mest inflytelserika högnivåspråken och kompilatorer finns för nästan alla plattformar. Det har också inspirerat många andra språk, som C++ och Java.

C++ Programspråk med stöd för dataabstraktion, objektorienterad programmering och generisk programmering. Språket utvecklades i början på 1980-talet av Bjarne Stroustrup vid Bell Labs. I dag är det ett av de populäraste programspråken och används inom allt från datorspel till konsumentelektronik. C++ är baserat på programspråket C och har anammat många begrepp och konstruktioner från det. Dock har man i C++ valt att stödja objektorienterad programmering genom att implementera bland annat klassbegreppet. C++ omfattar inte C i strikt mening då det finns flera skillnader i semantik mellan C och den delmängd av C++ som motsvarar C.

diffusion En egenskap som innebär att densiteten i en ruta i fluidberäknarens rutnät sprids ut till närliggande rutor. Det är en spridningsprocess som finns i naturen för att uppnå jämvikt i t.ex. luft och vatten.

Enum (enumererad typ) Ett set av namngivna värden som används för att få mer lättläst och lättare förstådd kod. Man deklarerar unika (värdena kan bara användas en gång för varje enum) heltalsvärden och namnger

dessa, skulle kunna vara t.ex. de olika färgerna i en kortlek (klöver, hjärter osv).

fluiddynamik Ett område inom fysiken som beskriver beteendet hos icke-fasta kroppar som vätskor och gaser.

FPS Står för "Frames Per Second" och är alltså antalet bilder som renderas per sekund. Låg FPS innebär att det ser ut som spelet hackar fram, medan med hög FPS så ser spelet ut att flyta på som vanligt. Ögat kan uppfatta upp till 70 FPS och allt över det är alltså onödigt.

funktionspekare Pekar till en minnesadress där det finns en funktion som tar de givna typerna och returnerar en specificerad typ, används för att förenkla kod så man kan välja olika funktioner beroende på olika kriterier.

inkompressibla fluider Fluider där densitetsvariationerna är försumbart små, gaser är exempel på inkompressibla fluider.

iOS iPhone OS, operativsystemen som alla iPhone, iPad och iPod touch använder. Finns i olika versioner.

kant Sammansättningen av två stycken hörn/punkter (linjen mellan dem).

konkav polygon En polygon där någon inre vinkel mellan två kanter överstiger 180 grader t.ex. stjärna eller H-formad polygon.

konservativt fält Vektorfält vars rotation är definierad och lika med noll.

konvex polygon En polygon där vinklarna mellan alla kanter är mindre än 180 grader t.ex. rektangel eller triangel.

lightmaps En teknik för att lägga till ljus vid rendering. Bygger på förrenderade texturer.

linje En linje är i detta sammanhang, en offset (avstånd från origo) och en riktning (en normalvektor). Linjen har oändlig utsträckning och man kan lätt m.h.a. linjär algebra kolla om något ligger framför eller bakom linjen.

matris Inom matematiken är en matris en rektangulär samling med tal.

normal En vektor vars riktning anger åt vilket håll något är orienterat. Har alltid längd 1 (normaliserad).

Objective-C Objektorienterad påbyggnad på vanliga C. Det går därför att blanda C och Objective-C relativt obehindrat i samma källkod.

OpenGL ES (OpenGL for Embedded Systems) Ett subset av OpenGL-biblioteket avsett för inbäddade enheter såsom mobiltelefoner, handdatorer och spelkonsoler.

polymorfism Ett sätt för att samla olika datatyper så man kan behandla alla dessa på samma sätt trots att de inte är helt lika.

projektionsmetoden Numerisk metod för att lösa tidsberoende problem som involverar inkompressibla fluider, som till exempel gaser.

shadow maps En teknik för att genera skuggor vid rendering. Scenen renderas i förväg från ljusets perspektiv och djupet för närmaste objekt sparas i en textur. Scenen renderas sedan ytterligare en gång ur vanligt perspektiv och ett värde läses ur texturen för att bestämma om objektet var i skugga ur ljusets perspektiv eller inte.

självadvektion Ett hastighetsfält som rör sig jämt med sig självt.

skalärprodukt Produkten av två vektorers belopp (längd) och cosinus av vinkeln mellan dessa.

solenoidalt fält Vektorfält vars divergens är definierad och lika med noll.

subversion (SVN) Versionshanteringssystem. Det håller reda på historik och förändringar och gör det möjligt att backa tillbaka i tiden eller att spåra vem som gjort vad.

trac Webbaserat projektledningss- och felrapporteringssystem med öppen källkod.

triangulisering Att utifrån skalet på en polygon dela upp denna så att den enbart består av enbart trianglar.

vektor I elementär matematik, fysik och teknik, är en vektor (kallas ibland euklidisk eller geometrisk) ett geometriskt objekt som har både en längd och riktning. Vår representation av en vektor innehåller endast en slutpunkt - vektorn antas utgå från origo.

viskositet En egenskap hos fluider som kan betecknas som en fluids tröghet eller inre motstånd, t.ex. har olja högre viskositet än vatten.

void-pekare Pekar till en plats i minnet, typen på datan som ligger på minnesadressen behöver inte vara specifierad i koden men kan kastas (konverteras) till någon typ (om de överensstämmer) vilket gör att man kan bygga koden väldigt generell och bara ha vissa kodbitar som är specifika.