Det får duga: Game Maker som prototypverktyg, och applicerad speldesign

Daniel Remar
a05danre@student.his.se
Handledare: Ulf Wilhelmsson

Institutionen för kommunikation och information
Examensarbete i medier: Dataspelsutveckling 30hp
C-nivå
Vårterminen 2008

Högskolan i Skövde
Box 408, 541 28 Skövde

Sammanfattning

Rapporten handlar om programmet Game Maker som prototypverktyg, och vänder sig till personer och företag som skulle kunna vara intresserade av detta. Syftet med arbetet var att skapa ett spel i Game Maker, "Ittle Dew", samt skriva om Game Makers egenskaper och egenheter. Spelet färdigställdes i tid, och rapporten tar även upp en mängd designfrågor som dök upp under skapandet av detta. Med hjälp av två brev från personer med erfarenhet av Game Maker som prototypverktyg och för skapandet av färdiga spel, och diskussion av dessa brev, hävdar jag till sist att programmet väl duger för prototyper, så länge användaren känner till de begränsningar som rapporten tar upp.

Nyckelord: Speldesign, Game Maker, prototyping, prototypverktyg, datorspel

Innehållsförteckning

1. Problemställning och mål

1.1 Syfte

Syftet med arbetet var att skapa ett spel i programmet Game Maker efter att ha planerat det tillsammans med Joel Nyström, motivera de olika designvalen i spelet, och använda det som en praktisk grund för den reflexiva rapporten. I den reflexiva rapporten byggs även en teoretisk grund runt Game Maker upp, och tillsammans utgör de båda ett kombinerat verk att basera slutdiskussionen på, i vilken jag tar upp Game Makers lämplighet som prototypverktyg för spel. Syftet med den reflexiva rapporten var även att skapa en slags kort introduktion för nybörjare till Game Maker, och peka ut dess likheter och skillnader med "traditionella" programmeringsspråk som C.

Syftet var dock inte att jämföra Game Maker med andra program som kan tjäna samma syfte. Det ligger utanför avgränsningen, och skulle inte vara fördelaktigt eftersom jag endast har erfarenhet med Game Maker.

När det gäller genusperspektiv på uppsatsen, anser jag att dess tekniska natur gör att det inte finns utrymme för en sådan diskussion. Jag har ingen insikt i t.ex. könsfördelningen i Game Makers användarbas, eller om någon sådan undersökning har gjorts.

1.2 Kontext och målgrupp

Inom spelproduktion anser jag att mycket möda kan besparas genom att först testa idéer i form av prototyper innan de implementeras i skarpt läge. Game Maker är ett verktyg som dels kan användas till icke-kommersiella spel, men på grund av dess egenskaper (mycket låg kostnad, snabb implementering, frihet och enkelhet i programmering) kan det även passa bra som prototypverktyg. Slutdiskussionen innehåller förutom egna reflektioner även två brev; ett från en anställd på ett företag som använt Game Maker för prototyper, samt ett från en medlem av ett team som gör komersiella Game Maker spel.

Rapporten riktar sig till fristående speldesigners, samt designers på kommersiella spelföretag, för att ge dem en insikt i programmet och om/hur det kan passa deras behov. Det är även möjligt att kapitlet som behandlar spelet som producerades under arbetets gång, Ittle Dew, kan ge inspiration till problemlösningar inom liknande spel.

Jag själv har fyra års erfarenhet av programmet, och har gjort spelen No Friction (2004), Hero (2005), Retrobattle (2005), Castle of Elite (2006) och Garden Gnome Carnage (2007), det sista vilket kom på andra plats i YoYoGames (www.yoyogames.com) vintertävling 2007. Spelen har laddats ned i sammanlagt cirka 25000 exemplar, och betydligt mer framgångsrika spelmakare inom Game Maker finns, vilket kommer tas upp i slutdiskussionen. Mina spel finns tillängliga på internet på min privata hemsida: http://www.remar.se/daniel

Läsaren antas ha en viss erfarenhet av spel, spelskapande och datorer.

2. Bakgrund och definitioner

2.1 Spel- och leveldesign

Speldesign definierar jag som skapandet av en spelidé, och tillräckligt med detaljrikedom i denna för att kunna producera ett spel av det. Leveldesign är i sin tur en del av speldesign, som går ut på att skapa den fysiska världen som ett spel utspelar sig i. Naturligtvis är leveldesign inte applicerbart på alla sorts spel, men i fallet med Ittle Dew kan det tydligt definieras som utformningen av de olika rummen spelaren tar sig igenom, vilka innehåller de olika föremålen och följer de regler som skapades i den tidigare speldesignfasen.

2.2 Skriptspråk

Enligt Ousterhout (1998) används skriptspråk för att programmera på en högre nivå än systemspråk ("system languages") som C och Java, d.v.s. längre bort från datorns rena maskinkod, och åstadkomma snabbare produktutveckling (min översättning och tolkning). Enligt Ousterhout används skriptspråk för att "limma ihop" komponenter av ett datorprogram, där varje komponent är skriven i systemspråk.

Ousterhout skriver att medan systemspråk gör det möjligt att skapa ett helt nytt program från grunden, förutsätter skriptspråk att det redan finns en grund av komponenter att arbeta mot. Med andra ord erbjuder skriptspråk en grund av små beståndsdelar att börja med, sedan kan utvecklaren snabbt skriva program som använder beståndsdelarna för att skapa mer komplexa system.

2.3 Prototyping

Prototyping kan innebära flera saker. Antingen är det ett konceptuellt "test" av ett program (eller en mindre del i programmet) för att avgöra hur väl en lösning fungerar, eller så kan ett program byggas upp av evolutionär prototyping där funktioner läggs till och tas bort tills programmet anses fullvärdigt, enligt Smith (1991). I det första fallet är det ett sätt att förbereda skapandet av ett program, och eventuellt testa det på personer som kommer att använda det färdiga programmet. Enligt Smith (1991) kan prototyping bland annat vara ett sätt att låta skaparen av ett program ändra sina planer mitt i utvecklingen, något som inte kommer att märkas när programmet väl skapas på riktigt efter den fullbordade prototypen, och som inte kostar onödig utvecklingstid.

I fallet med Ittle Dew, spelet som jag och Joel designade för detta projekt, var det gjort för att vissa saker skulle kunna ändras under utvecklingen, inklusive att lägga till och ta bort betydande spelfunktioner. Mycket av spelmekaniken lämnades att utvecklas under vägens gång.

2.4 Game Maker

Game Maker är ett program för Windows skrivet av Mark Overmars. Programmet sköts och uppdateras numera av YoYoGames (www.yoyogames.com). Dess slogan är "Easy game development" - programmet hanterar grafik, ljud och objekt, och kan programmeras med ”drag-and-drop”, en slags metaprogrammering, vilket gör det snabbt att skapa enkla spel. Med det inbyggda GML-skriptspråket kan även ”mer traditionell” kod skrivas för att utnyttja de funktioner som inte finns som ”drag-and-drop”, och göra spelen betydligt mer avancerade. Hur allt detta fungerar kommer att tas upp i större detalj i kapitel 4.

Game Maker är främst gjort för 2D-spel, och är ett relativt kraftfullt verktyg när det används korrekt; för exempel se spel som *The Cleaner* ("darthlupi" 2006) och *Return to Sector 9* (Chris Roper 2007). Dessa spel finns att hämta på www.yoyogames.com. Det är möjligt i GM6.0 och framåt att implementera enklare 3D-grafik, men det var enligt programmets manual inte Gamer Makers främsta användningsområde. Dessa paragrafer står att finna i manualen till Game Maker 7:

> The 3D functionality in *Game Maker* is limited to the graphics part. There is no support for other 3D functionality. Once you start using 3D graphics you might get problems with other aspects of *Game Maker*, like the views, depth ordering, etc. The functionality is limited and has low priority to be extended. So don't expect support for 3D object models, etc.
>
> The 3D functions in *Game Maker* can be used to make some nice 3D games. However, they are limited in functionality and still leave quite a lot of work to you. Don't expect that you can make your own *Quake* with it. *Game Maker* is and remains primarily a package for making 2-dimensional games.

3. Ittle Dew - ett Game Maker spel i praktiken

3.1 Introduktion till verket

3.1.1 Samarbetet med Joel

I början av arbetet hittade jag och Joel Nyström (hädanefter kallat "vi") på fyra spelidéer, med målet att välja ut den med mest potential för att bli ett intressant och varierat spel, som vore möjligt för mig att ensam implementera. Den första idén var ett pusselspel där blocken kunde ha tre attribut som kunde kombineras. Det andra var ett äventyrsspel som utspelar sig i en borg, med fyra huvudföremål och deras inbördes relationer. Den tredje idén var ett actionspel där en tuff actionhjälte spränger sig igenom spelet och sprider skrot och skräp överallt, medan hans mamma måste städa upp efter honom. Den fjärde idén var ett strikt pusselspel där spelaren styr ett block runt en labyrint, som kan klistra fast andra block på sig självt och ta sig till mål.

Vi bestämde oss för äventyrsspelet "Ittle Dew" med fyra föremål, vars grund var väldigt likt The Legend of Zelda (Nintendo 1987) till Nintendo Entertainment System. Vi ansåg att det hade bäst potential, eftersom vi bara stötte på problem i gameplay-designen till de andra spelen.

3.1.2 Målen med spelet

Spelet skulle kunna implementeras av mig inom loppet av ett par månader, och ha en leveldesign som jag inte behövde bry mig särskilt mycket om, eftersom det var Joels arbete. Implementationen skulle göra det så enkelt som möjligt för Joel att bygga sina banor, och diverse spelelement skulle dynamiskt kunna anpassa sig efter bandesignen, t.ex. dörrar och element som aktiverar dem. Fiender skulle kunna placeras ut i ett rum och sedan bete sig korrekt, och så vidare.

Spelet skulle vara tillräckligt speltekniskt djupt för att vi skulle stöta på designfrågor under produktion, och det skulle vara möjligt att implementera spelet lite hur som helst, för att utnyttja Game Maker till fullo.

3.1.3 En kort genomgång av spelet

Ittle Dew är en äventyrare vars uppgift är att komma till det översta rummet i en stor borg. Hur hon tar sig dit kan variera, och alla verktyg hon hittar behövs inte för att klara spelet, men det finns många hemligheter att hitta. Se Fig 1 för en typisk scen i spelet.

Spelaren använder i huvudsak fyra föremål för att lösa pussel och bekämpa fiender i realtid (föremålen kan också användas i kombination med varandra på olika sätt, se avsnitt 3.2.2):

- Ett närstridsvapen, vilket börjar som en pinne, uppgraderas till ett magiskt eldsvärd och till sist blir en eldkastare. Om spelaren tar uppgraderingarna i omvänd ordning erhåller hon däremot ett gevär som istället för hagel skjuter kaskader av brinnande explosiva motorsågar.
- Bomber med stubiner, vilka expldoerar en kort stund efter att de placerats.
- Ett isvapen, som kan frysa diverse föremål.
- En teleporteringsstav, vilken kan förflytta alla möjliga föremål och fiender runt i rummet, samt Ittle själv.

Spelaren rör sig från rum till rum, vilka har olika antal utgångar till andra rum, och utforskar slottet med verktygen hon hittar för att till sist ta sig till trappan högst upp. På vägen stöter hon på monster som måste besegras eller undvikas, och ibland användas till att lösa pussel. Pusslen är oftast rutnätsbaserade, d.v.s. knuffbara block kan endast knuffas i korta steg, som om de rörde sig längst ett rutnät.

Spelets designdokument är bifogat som bilaga A. Det beskriver spelets innehåll i form av punkter och korta beskrivningar, och kan vara intressant för att se spelets grundstenar och jämföra dem med det färdiga spelet. Märk dock att designdokumentet var skrivet för mig och Joel, så mycket av spelets struktur var implicit och behandlas inte i dokumentet.

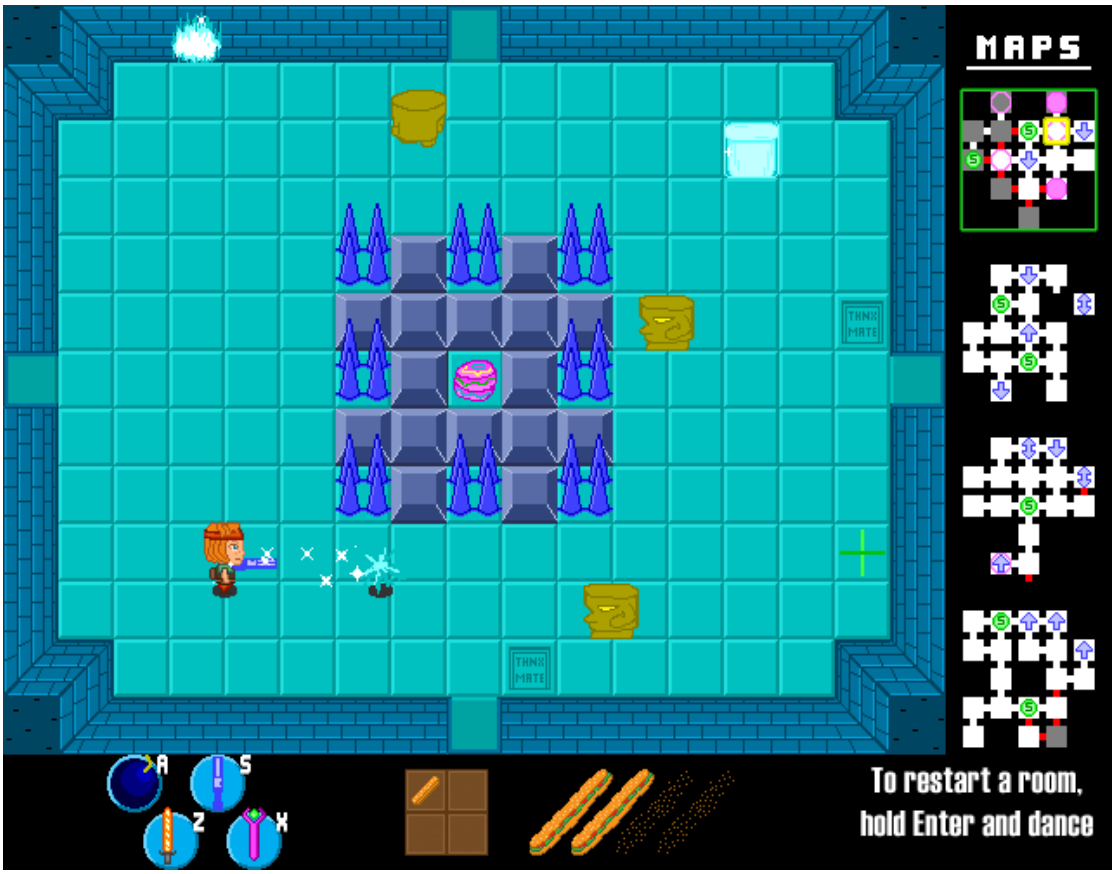

Fig 1. En scen från Ittle Dew. Nederst till vänster syns bomber, isvapnet, eldsvärdet och teleporteringsstaven, och i mitten av skärmen en piñata

3.2 Konceptfasen - designval innan produktion

3.2.1 Vad händer om spelaren fastnar i ett rum?

Som spelet fungerar är det möjligt för spelaren att spärra in sig själv i ett rum, och inte kunna komma ut. Istället för att behöva planera rummen så att detta aldrig kan hända, vilket vore svårt och tidskrävande, kom vi på idén om att ha en återställningsfunktion. Spelaren kan återställa ett rum som det var när hon kom in, och karaktären flyttas tillbaka till dörren hon kom igenom. Eftersom detta kan missbrukas genom att flytta karaktären bort från farliga fiender, ville vi lägga en viss restriktion på återställningsfunktionen.

En idé var att bestraffa spelaren, genom att ta hälsa från karaktären varje gång spelaren lyckats fastna i ett rum. Problemet med detta var att spelaren kanske inte alls var i fara från monster, utan bara fastnat bakom några block. Ett annat alternativ var att karaktären måste stå still och ladda upp återställningsfunktionen i några sekunder, utan att det kostar något. Då kunde spelaren inte utnyttja det, eftersom karaktären var sårbar medan återställningsfunktionen laddas upp. Fördelen med detta var att spelaren inte känner sig bestraffad, men nackdelen var att det inte är lika snabbt som den första metoden.

3.2.2 I vilken ordning skall spelaren hitta verktyg?

Vi ville att varje verktyg spelaren hittar skulle fungera väl tillsammans med alla andra hon redan hittat, eller öppnar upp nya möjligheter. Till exempel finns bomber med stubiner utplacerade i vissa rum, som dels kan flyttas runt med teleporteringsstaven och dels tändas med en brinnande pinne. Det senare förutsätter att det finns en fackla i närheten att tända pinnen på. När spelaren väl hittar eldsvärdet, vilket fungerar som en konstant brinnande pinne, behövs inte längre en fackla för att tända bomber. När spelaren sedan hittar sina egna bomber att placera ut, kan de användas i rum som annars inte har bomber utplacerade i förväg. Det vore alltså logiskt att inte först ge spelaren bomberna, utan ta ett steg i taget så att alla möjligheter inte öppnas upp på en gång.

Vidare finns "övermäktiga" verktyg, föremål som gör pussel och strider väldigt enkla och får spelaren att känna sig mäktig. Dessa var tänkta att utgöra en belöning till spelare som utför "sequence breaks", d.v.s. använder sin skicklighet för att ta verktygen i spelet i en annan ordning än den omedelbart uppenbara. Exempelvis var det tänkt att spelaren först skall hitta eldsvärdet, och sedan eldkastaren, eftersom den senare har samma attribut men även kan attackera på avstånd. Om spelaren dock hittar eldkastaren först, och sedan går tillbaka för att hämta eldsvärdet finns ett annat vapen där istället; det brinnande motorsågs-skjutande geväret. Speltestare har påpekat att användandet av detta verktyg står i så stark kontrast till resten av spelet, att det blir roligt att använda.

3.3 Planeringen av produktionen - att plocka här och där

För att utnyttja att jag nu använde ett verktyg gjort för att snabbt skapa enkla spel, bestämde vi oss för att använda följande arbetsmetod. Jag implementerar inte spelet som

en programmerare skulle på ett företag, där allt planeras och systematiskt implementeras i logiska steg, utan plockar mest fritt från ett dokument där alla beståndsdelar av spelet är kort beskrivna. Detta gör att många olika designproblem kan komma i dagen mellan beståndsdelar som annars implementerats mycket sent i utvecklingen, och vi kan förhindra misstag och ofullständiga koncept. I ett spel så relativt komplicerat som Ittle Dew står det snart klart att utan någon form av prototyping skulle det vara mycket svårt att undvika denna sorts oförutsedda designfrågor. Detta anser jag är en av de starkaste argumenten för prototyping, vilket jag kommer att återkomma till i slutdiskussionen.

3.4 Nya eller oväntade designval under produktion

3.4.1 Dörrhalvor och när en dörr blivit permanent öppnad

I spelet går spelaren från rum till rum med hjälp av dörrar. Dessa kan vara stängda och öppnas då med "triggers", t.ex. golvplattor, s.k. "pressure pads". Frågan är hur det går till när spelaren öppnar en dörr från ena sidan och går in i det andra rummet; ska dörren då stängas igen för att åter behöva öppnas inifrån det nya rummet spelaren gått in i? Vi bestämde oss för att en passerad dörr skulle förbli permanent öppnad, vilket även gör så att spelaren inte behöver lösa samma pussel varje gång hon ska passera rummet. Det kan också kännas jobbigt för spelaren om en dörr hon precis lyckats öppna stängs så fort hon passerar den, och inte låter henne gå tillbaka samma väg hon kom.

Nästa fråga är när en dörr räknas som permanent öppnad. I ett rum kan det finnas fler än en trigger som leder till dörren, och alla måste aktiveras samtidigt för att öppna den. Antingen låter jag då dörren öppnas när alla är aktiverade och stängas så fort de inte är det, och låta dörren förbli öppen så fort spelaren passerat den för första gången. Då uppstår dock problemet att spelaren kanske löser pusslet, men lämnar rummet genom vägen hon kom och senare kommer tillbaka, varpå dörren åter stängts. För att spelaren inte ska känna att spelet motarbetar henne, valde vi att låta en dörr bli permanent öppen så fort den öppnats med sina triggers, utan att spelaren behöver passera den. Detta innebär i förlängningen att alla triggers som leder till samma dörr på andra sidan automatiskt blir aktiverade och räknas som "använda", och går ej att aktivera.

3.4.2 Vad återställs när spelaren lämnar ett rum?

För att inte låta spelaren göra pussel olösliga, och utan att behöva använda återställningsfunktionen hela tiden, måste pusselelement inom rummet som block, triggers och pelare återställas när spelaren lämnar rummet. Detta är logiskt på det sätt att eftersom varje pussel som involverar dessa element endast håller sig inom ett rum, och eftersom spelaren bara ser ett rum i taget, kan återställningen ske utan att det känns märkligt för spelaren. När det gäller fienderna blir saken svårare; när ska de återställas? Spelaren vill inte nödvändigtvis behöva kämpa sig igenom ett actionfyllt rum varje gång hon passerar det, så fienderna borde förbli besegrade.

Vissa fiender kan dock behövas för att lösa pussel i rummet, dessa typer borde då återuppstå när spelaren lämnar rummet och kommer tillbaka. För att spelaren inte ska bli utmattad av dessa fiender, borde de inte heller vara lika svåra som de som inte krävs för att lösa pussel.

En tredje indelning av fiender dök dock upp under implementationen; de räknas som action-fiender, men kan ändå användas till pusslande. Vi bestämde oss för att låta dessa fiender vara, och endast använda deras pusselfunktioner till "sequence breaking", då det krävs mer skicklighet och planering för att göra detta. Vi bestämde också att alla fiender återuppstår varje gång spelaren rör sig mellan två plan i borgen, detta för att borgen inte ska bli helt ödslig efter ett tag.

3.4.3 Teleporteringsstaven blev övermäktig

Teleporteringsstaven kunde från början flytta på fiender, spelaren, vissa projektiler och även knuffbara block. När allt detta implementerades visade det sig dock att förmågan att kunna teleportera knuffbara block gjorde pussel som involverade dem alldeles för enkla - spelaren kunde omarrangera blocken i rummet precis hur hon ville. Detta motarbetade tanken om att låta teleporteringsstaven bli ett av de första verktyg vi ville att spelaren skulle hitta (eftersom det är det mest unika verktyget, och kan användas tillsammans med flest andra föremål). Vi bestämde då att ta bort förmågan att teleportera knuffbara block, och istället låta spelaren hitta en uppgradering till teleporteringsstaven som möjliggör detta senare. Denna uppgradering hittas lämpligen allra sist i borgen; kanske behövs den inte ens för att klara spelet.

3.4.4 Piñatas dödar alla fiender i rummet

När spelaren plockar upp ett nytt föremål ur en piñata (piñatas är spelets motsvarighet till de traditionella skattkistorna, där skatter och verktyg hittas), stannar karaktären upp och ett stycke text meddelar spelaren om vad hon hittat och hur det används. För att undvika att fienderna attackerar karaktären medan hon stannar upp finns en mängd lösningar. Från början blev karaktären helt enkelt odödlig, men då uppstod problemet att fienderna flockades runt henne, och spelaren fick vänta på en öppning att fortsätta spela för att inte bli skadad så fort hon återfick kontrollen. Fienderna kunde istället stå stilla medan karaktären gjorde det, men idén om att alla fiender i rummet istället skulle besegras så fort spelaren plockar upp föremålet valdes för att det ger lite mer spänning. Första gången spelaren kommer in i ett svårt rum kan hon sikta mot piñatan för att snabbt ta hand om fienderna, men följande gånger kommer piñatan inte att finnas kvar. Detta gör inte bara så att spelaren inte behöver ta itu med svårare actionrum på allvar förens hon kommer tillbaka till det senare i spelet, hon kan även undvika att plocka upp föremålet i piñatan om hon vill spara det just för att ta hand om fienderna.

3.4.5 Glidande isblock och pressure pads

När spelaren knuffar ett isblock glider det längs marken tills det krockar med något och stannar. Eftersom pusslen är rutnätsbaserade stannar isblocket så fort det finns ett

blockerande föremål någonstans i den ruta som blocket var på väg att glida in i. Visserligen ser det lite konstigt ut, men det är snarare en fråga om spelmekanik och funktionalitet.

Pressure pads är plattor som sjunker ned i golvet när något tungt står på dem, som knuffbara block och isblock (Fig 2). När ett antal pressure pads har tryckts ned samtidigt öppnas en dörr, eller pelare sänks ned. Frågan är vad som händer när ett isblock kommer glidande och bara tillfälligt passerar över en pressure pad; ska den tryckas ned tillfälligt eller inte alls? Vi valde det senare, och bestämde att isblocket bara skulle trycka ned plattan om blocket inte var i rörelse, annars kunde problem uppstå om spelaren antog att hon var tvungen att lösa ett pussel genom precis timing i knuffningen av flera isblock.

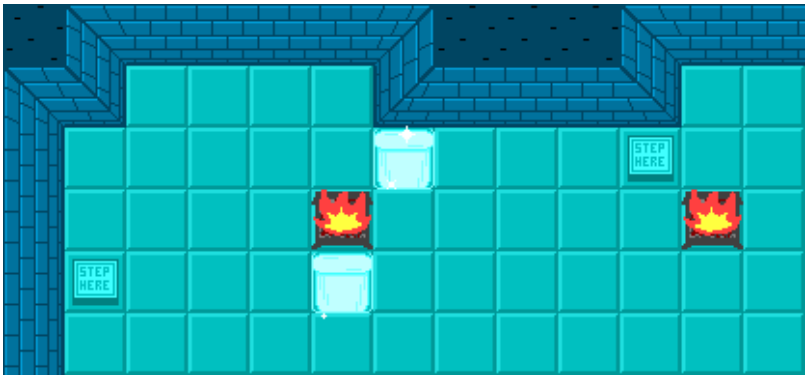

Fig 2. Isblock, pressure pads och muntert brinnande facklor. Om det nedre isblocket knuffats åt vänster, hade det stannat mot den vänstra väggen och tyngt ned en pressure pad. Om det övre blocket knuffats åt höger hade det däremot glidit rakt över sin pressure pad och inte aktiverat den.

3.4.6 Logiken bakom facklorna

Facklor är en speciell sorts trigger som både kan tändas och släckas med rätt verktyg. Till skillnad mot andra triggers kan de börja som antingen tända eller släckta, och ett pussel kan lösas genom att tända de som från början är släckta, eller vice versa.

Det första problemet är hur spelaren ska veta om en fackla ska tändas eller släckas för att lösa pusslet. Vissa facklor ska också inte ha med pussel att göra. Vi löste detta genom att göra tre sorters facklor: en grå, som inte kan trigga någonting; en blå, som börjar tänd men ska släckas; och en röd, som börjar släckt och ska tändas. Spelaren ser nu vilken funktion varje fackla har. Självklart måste deras funktion introduceras ordentligt i spelet först; detta är upp till leveldesignern.

Det andra problemet är huruvida spelaren fortfarande kan tända och släcka facklorna efter att de använts för att lösa pusslet, och vad som då händer med dörren de just öppnat. Eftersom vi redan fastställt att öppnade dörrar ska förbli öppna, satte vi en lampa på varje fackelstod. Dessa tänds så fort facklorna använts för att lösa pusslet, och förblir tända för att indikera att pusslet är löst och inte behöver lösas igen. Facklorna kan fortfarande tändas och släckas för att det ska kännas logiskt för spelaren.

3.5 Det färdiga spelet: en kort diskussion

Implementeringen gick mestadels enligt planerna, och orsakade inga oöverkomliga programmeringsproblem trots spelets komplexitet. Jag stötte dock på ett par problem i kollisionshanteringen, vilket jag skrivit mer om i avsnitt *4.3.1. Collision event kontra place_meeting()*. Utöver detta utgjorde Game Maker inte några större begränsningar i vad som var möjligt i vår design av spelet, och det hela var gjort på mindre än 6 veckor.

I slutändan blev spelet väldigt likt The Legend of Zelda (Nintendo 1987), med verktyg som kunde interagera med varandra på ett så stort antal sätt att speltestare ofta fastnade och fick fundera ett tag innan de hunnit testa alla möjliga samband och effekter. Det är möjligt att spelet till slut blev för avancerat - ett exempel är ett pussel där spelaren måste placera en "portal" över ett spikgolv, frysa en vägg med isvapnet, och sedan skjuta först en teleporteringsboll mot isväggen så att den studsar tillbaka, och därefter skjuta på den flygande teleportbollen med en eldkastare, vilket förflyttar eldsflamman från eldkastaren till den förstnämnda "portalen" så att den träffar en fackla och tänder den. Detta låter förmodligen lika förvirrande och svårt att visualisera för läsaren av denna rapport som för spelaren av spelet. Spelaren har nu i korthet använt tre olika verktyg för att "skruva" en eldsflamma från eldkastaren 90 grader för att komma åt facklan som skulle tändas. Andra pussel involverar att placera ut en tänd bomb med en stubin, frysa bomben med isvapnet, och sedan teleportera runt bomben i rummet tills den hamnar på rätt plats, tinar, och träffar en "kristalltrigger" med explosionen för att öppna en dörr. Även om spelaren har fått lära sig alla dessa detaljer en i taget under spelets gång, tvingar slutfasen av spelet henne att återanvända allting hon lärt sig tillsammans, i flera stora pussel.

Jag hoppas dock att för en spelare som är intresserade av djupt logiska pussel, fyller spelet sitt syfte och erbjuder en ordentlig utmaning.

4. På djupet om Game Maker, begränsningar och lösningar

4.1 Introduktion till programmering och Game Maker

4.1.1 Kompabilitet

Game Maker version 5.3A är kompatibelt med Windows 95 och uppåt, medan Game Maker 6.x är inkompatibelt med Windows Vista. Game Maker 7 är endast kompatibelt med Windows XP och uppåt. Det finns dock ett litet program på www.yoyogames.com som konverterar ett kompilerat GM6.x spel till GM7 så att det blir Vista-kompatibelt.

I skrivande stund stöder YoYoGames inte längre några versioner av programmet lägre än GM7, och gamla licensnycklar kan inte heller köpas, vilket innebär problem för de som vill maximera kompabiliteten i sina spel eller använda de tidigare GM-versionerna.

Game Makers funktionalitet kan utökas med DLLs, vilket kan möjliggöra skapandet av mer avancerade spelprototyper, men jag har valt att inte ta upp de befintliga DLLs i detta arbete.

4.1.2 Drag and Drop och pseudokod

Schildt (2000) förklarar i *Chapter 27: Software Engineering Using C* pseudokod genom exempel, där programmeraren inte skriver ren kod som är tänkt att förstås av en dator, utan ord och meningar i klartext, arrangerade i samma logiska ordning som programmet ska utföra dem. Pseudokod är ett sätt att designa ett program utan att behöva tänka på hur det skulle se ut i kod.

Game Maker har två sätt att programmera; antingen i ren kod via det inbyggda språket GML (Fig 4), eller genom "Drag and Drop" (Fig 3), en slags pseudokod-programmering som förstås av Game Maker. Med Drag and Drop väljer spelmakaren först en handling, t.ex. *Create event, Step event* eller *Collision event* i vilken koden komemr att utföras, och placerar därefter ut symboler som representerar kod, och sätter enkla värden i dem. Exempel på symboler är t.ex. *Start moving in a direction*, där spelmakaren väljer en av åtta riktningar och symbolen är åtta pilar som utgår från en mittpunkt, eller *Play a sound* där ett ljud väljs från resurslistan och symbolen ser ut som en högtalare.

Notis: GML-kod skrivs med hjälp av Drag and Drop symbolen *Execute a piece of code*, eller genom att skapa ett "script" och exekvera det antingen genom GML-kod eller från Drag and Drop symbolen *Execute a script*.

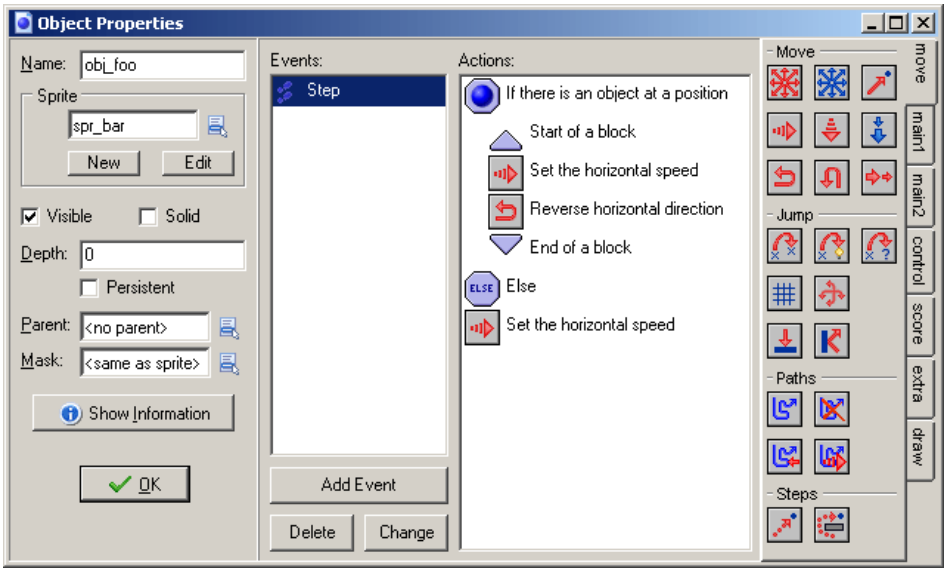

Fig 3. Exempel på Drag and Drop ikoner (under rubriken "Actions")

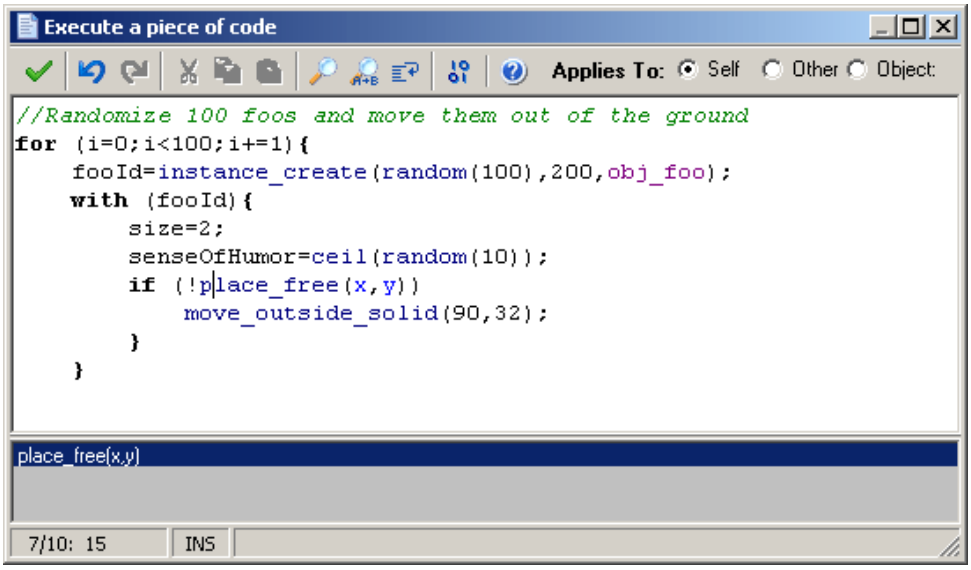

Fig 4. Exempel på GML-kod. Här utnyttjas faktumet att varje instans har ett unikt ID, vilket sparas undan i variabeln "fooId" och används för att anropa instansen och utföra kod med den.

Fördelen med Drag and Drop är att det går fort att programmera i det, och är enkelt att läsa och förstå för en nybörjare. Nackdelarna är dock flera:
- Det går inte att använda de flesta funktioner som finns tillgängliga i GML-kod, eller programmera på en liknande avancerad nivå.
- Det blir svårt att läsa koden om den blir för lång, eftersom symbolerna tar stor plats i gränssnittet, och inte alltid framför logisk information.
- Enligt egen erfarenhet i att konvertera ett stort projekt från Drag and Drop till GML, konstaterade jag att Drag and Drop tar dubbelt så stor plats i projektets filstorlek som skriven GML-kod och verkar något långsammare att utföra i körtid.

Jag rekommenderar att nybörjare till programmet börjar med att programmera i Drag and Drop, medan en van programmerare troligen hellre börjar direkt med GML och läser manualen för att lära sig språkets syntax.

4.1.3 Kompilering kontra tolkning

Enligt Schildt (2000) *Chapter 1: An Overview of C* finns det två generella sätt att exekvera datorprogram, genom kompilering eller tolkning ("interpreting"). En tolk läser källkoden en rad i taget och utför vad som står där. I fallet med Java konverteras källkoden först till en mellanform som sedan tolkas, men i vilket fall krävs en tolk för att kunna köra programmet. En kompilator, å andra sidan, läser hela programmet och konverterar det till maskinkod, ett språk som datorn direkt förstår utan att behöva en tolk. Tolkade program är i regel långsammare i körtid än kompilerade, ungefär som kommunikation mellan två människor som talar olika språk med en tolk emellan sig är långsammare än om båda talat samma språk, med fördelen att den ena inte först behöver lära sig den andres språk (min metafor).

Game Maker är ett tolkat programmeringsspråk, vilket gör det långsamt och krävande i körtid, men det går fort att "kompilera" och starta ett spel. Trots att en användare kan konvertera ett Game Maker spel till en självständigt körbar EXE-fil, vilken inte kräver Game Maker för att spela, kompileras spelet inte egentligen till maskinkod. Med rätt verktyg kan en Game Maker EXE-fil konverteras tillbaka till läsbar och editerbar kod (vilket dock är olagligt).

Att Game Maker är ett tolkat språk gör det enklare att felsöka ett spel i körtid med debug-läget, eftersom användaren kan ändra värdet på alla variabler i spelet och exekvera ny kod i körtid. Det möjliggör också en funktion som kallas *execute_string()*, vilken likt debug-mode låter användaren skapa och exkvera "ny kod i koden" - med andra ord kan användaren skriva kod som dynamiskt sätter ihop en textsträng i körtid, och säga åt Game Maker att exekvera textsträngen som om det vore ny kod. Detta rekommenderas dock inte till seriösa projekt, eftersom det inte är logiskt nödvändigt och allvarliga fel kan uppstå; användandet bör begränsas till felsökning.

4.1.4 Total öppenhet i variabler

Schildt (2000) skriver i *Chapter 27: Software Engineering Using C* att en viktig del i att skapa felsäkra program är att försäkra sig om att varje del av programmet "döljer sin kod" för alla andra delar som inte behöver känna till den. Kort sagt är det som att berätta en hemlighet endast till dem som verkligen behöver veta den; att inte låta en funktion dela med sig av sin data till några funktioner som inte behöver den. Schildt avråder från användningen av "global data", där variabler inte är specifika till någon funktion eller något objekt.

Game Maker struntar blankt i allt detta. Alla instanser i ett Game Maker spel kan när som helst läsa och skriva till alla andra aktiva instansers personliga variabler, så länge de känner till deras unika identifierare. Om ett objekt ”obj” bara har en instans, som har

variabeln ”var”, räcker det med att skriva ”obj.var” vart som helst, i vilket objekt som helst, för att få tillgång till denna variabel. Om en specifik instans sökes finns det flera sätt att hitta den – exempelvis kan funktionen ”with(obj)” användas, vilket utför kod med alla instanser av typen ”obj”. Deras koordinater och diverse variabler kan då avläsas och jämföras för att hitta den önskade instansen.

Det finns även globala variabler, anropade genom att skriva ”global.” framför dem, som inte tillhör någon specifik instans och behålls även om spelet byter rum. Det blir snabbt farligt om instanser blir beroende av varandras personliga variabler, eller delade globala variabler, men fördelen är att enkla lösningar och korsverkanden kan skrivas väldigt snabbt. I Ittle Dew används ett osynligt kontrollobjekt i varje rum, som håller koll på allting som inte tillhör någon specifik instans, och exempelvis ritar ut HUDen. (Detta objekt har även lägst *depth* i spelet för att hamna sist i ritordningen, så att ingenting ritas ut framför HUDen; se nästa avsnitt.)

4.1.5 Frame-baserad motor

Ett spel i Game Maker körs i ett visst antal "frames", eller grovt sett "bilder" per sekund. Antal frames per sekund ställs in i ett rums egenskaper eller i körtid genom GML-kod. Varje frame kan omöjligen övergå till nästa förrän alla aktuella operationer beräknats - med andra ord, om en väldigt krävande operation utförs på frame 1, kommer spelet att helt stanna upp tills operationen beräknats färdigt och spelet kan gå vidare till frame 2. Ju mer krävande ett spel blir, och ju fler instanser som finns ute i spelrummet, desto långsammare kan det gå.

Utritning sker också normalt endast i slutet av varje frame, kallat *Draw* (detta kan kontrolleras med GML-kod). Genom att placera kod i *Step event* för ett objekt kan användaren bestämma vad objektet ska utföra varje enskild frame, och vet med säkerhet att om hastigheten exempelvis sätts till en "pixel" per frame, kommer objektet alltid röra sig så fort utan undantag. Om ingen slumpfunktion används är ett Game Maker-spel av naturen deterministiskt, d.v.s. kommer att fungera exakt likadant varje gång p.g.a. dess frame-baserade motor. Varje objekt har också ett djup, eller *depth*, vilket bestämmer dess plats i ritordningen. Ju lägre djup, desto senare kommer dess handlingar att utföras varje frame, och desto närmare skärmen kommer det att ritas ut. Depth kan även ändras i körtid med kod. Bakgrundslager och ”tile”-lager (se avsnitt 4.4.3) har också sina egna djup - dessa fungerar likadant som djupet hos objekt.

Ett exempel på logiska fel som kan uppstå är om en ny instans skapas i ett objekts *Draw event*. Eftersom *Draw* är det sista som händer varje frame, kommer den nya instansen att placeras ut, och därefter utföra alla sina handlingar, inklusive sin egen *Draw*. Detta innebär att instansens grafik kommer att ritas ut överst den framen, oavsett *depth*. En logisk kullerbytta kan även göras här; om ett objekt skapar en instans av sig själv i sin *Draw event* eller *Create event*, kommer spelet att låsas eftersom dessa beräkningar logiskt sett kommer att ta oändligt lång tid. Tyvärr kan ett Game Maker spel som fastnat i en sådan "oändlig loop" inte avbrytas på något annat sätt än att tvinga det genom Windows Task Manager.

4.2 Versionsoberoende leveldesign

4.2.1 Game Makers versionsberoenden

I detta kapitel innebär ordet "version" de olika versionerna av ett spelprojekt, t.ex. varje gång något nytt implementeras; det innebär inte de olika versionerna av Game Maker självt. Med andra ord handlar kapitlet inte om att göra bandesignen kompatibel mellan Game Maker 5 och 6, utan mellan olika versioner av ett spel som utvecklas i en och samma Game Maker version.

Game Maker var troligtvis inte gjort för att flera personer ska kunna jobba på samma projekt. Alla resurser som skapas i spelet har en unik identifierare, vilket innebär att två resurser kan ha samma namn. Det finns stöd för att slå ihop två projekt, men inga resurser kommer då att skrivas över - om de två projektfilerna har en identisk uppsättning resurser, kommer resultatet bli ett projekt med en dubbel uppsättning av allting. Objekt i det ena spelet kan inte peka på resurser i det andra, om användaren inte manuellt ändrar på detta, inte heller kan instanser från ett projekt placeras ut i ett annat med Game Makers interna rumeditor. Detta innebar problem för vårat spel eftersom Joel skulle bygga banorna och jag implementerade spelet. Om resurser har samma namn kan även förvirring snabbt uppstå. Det finns dock ett undantag; om en resurs anropas med dess namn via GML-kod, och flera har samma namn, kommer Game Maker att använda resursen med lägst nummer på sin identifierare - detta möjliggör ett kryphål i logiken, vilket ger upphov till vår idé om versionsoberoende leveldesign.

4.2.2 Versionsoberoende leveldesign

För att jag inte skulle behöva spendera lång tid på att skriva en extern levelleditor, som skrev och läste från textfiler och på så sätt var versionsoberoende, bestämde vi oss för att låta Joel bygga banorna i Game Makers interna editor och sedan slå ihop detta projekt med min senaste version av spelet. Vi löste detta genom en process vi kallar *ghosting*. Joel sätter ut ghost-instanser i sin editor, objekt som egentligen bara består av enkla bilder för att han ska kunna se vilket objekt det är. Spelet använder sedan en lista för att ersätta dessa objekt mot de riktiga objekten när det körs, genom att anropa de riktiga objekten med deras namn och ta bort ghost-objekten. Eventuella variabler som lagts in i de individuella instanserna förs också över. Självklart måste alla objekt, deras funktion i spelet samt deras variabler ha bestämts under designfasen av spelet, så att det inte blir några missförstånd. När projekten slås ihop behöver dock en viss kort process utföras; eftersom dubbletter av alla objekt följer med, behöver alla resurser utom ghost-objekten, deras grafik och "banbyggarversionens" rum tas bort från "banbyggarversionen" efter ihopslagningen. "Spelversionen" innehåller alltså all logik och färdiga resurser, medan banbyggarversionen inte behöver veta någonting om dem. Detta är just tanken med ghosting-processen, likt ett datorprogram som anropar en funktion inte behöver veta hur den internt utförs.

En nackdel med ghosting är att det tar någon minut att slå ihop banbyggarversionen med spelversionen. Spelmakarna måste vara noga med att alltid använda de senaste

uppdateringarna av dessa, och att aldrig implementera nya saker i banbyggarversionen, annars kan det snabbt bli förvirrande. En annan nackdel är att det tar en stund för spelet att byta ut ghost-objekten mot de riktiga objekten i körtid. Det avrådes att använda mer än ett per tusen objekt per ”rum”, eftersom det är krävande för Game Maker att förstöra och skapa ett så stort antal instanser på en gång.

Fördelen är ett effektivt sätt för två personer att arbeta på samma projekt, där den ena ansvarar för banbyggandet. Naturligtvis kan det även tas ett steg längre – ett antal personer kan bygga en bana var, så länge det står klart vilka banor som ska behållas och vilka som ska tas bort under ihopslagningsprocessen. Det är fördelaktigt att tidigt göra en checklista på hur ihopslagningen kommer att gå till varje gång.

4.3 Game Makers egenheter

4.3.1 Rum

Ett Game Maker spel utspelar sig i ett eller flera rum (rooms). Ett rum innehåller instanser av spelets objekt, och inställningar som storlek och fördefinierade bakgrunder. Spelet kan hoppa från rum till rum, men endast globala variabler förs över mellan rummen - allt annat återställs. Det är möjligt att göra ett rum "persistent", vilket återställer dess status när spelet hoppar tillbaka dit, men det kan orsaka problem och vara betydligt mer krävande.

Varje plan i slottet i Ittle Dew är ett eget rum. När spelaren går upp och nedför trapporna byter spelet rum.

4.3.2 Collision Event kontra place_meeting()

Om två objekt ska kunna kollidera, kan det antingen kollas genom Game Makers *collision event (Fig 5)*, eller så används kod-funktionen *place_meeting()* (Fig 6) i *Step event*. De båda fungerar dock olika. När *collision event* mellan två objekt utförs, har objektet som utförde kollen tillgång till den unika identifieraren på instansen den kolliderade med, och kan då enkelt läsa och skriva värden till denna genom att använda nyckelordet "other". Däremot finns ingen motsvarighet i kod - *place_meeting()* returnerar inte vilken instans som kolliderades med, endast om en kollision inträffade eller inte.

Det finns tillfällen då *collision event* inte är att föredra, främst när solida objekt är inblandade. Objekt med soliditet är ett specialfall; när *collision event* utförs mellan två objekt, och minst ett av dem har solid-flaggan, kommer deras positioner att återställas till vad de var föregående frame. Detta innebär problem om spelmakaren vill att ett objekt ska kunna passera fritt genom ett solitt objekt, och kod måste istället användas, eftersom *place_meeting()* av någon anledning inte återställer positionerna. För att göra saken krångligare finns även funktionen *place_free()*, vilken kollar endast mot solida objekt - normalt väggar, hinder osv. Om spelmakaren vill att ett objekt ska kollidera mot de flesta

solida objekt, men inte alla, måste specialfall skrivas i kod. Soliditet och kollisioner orsakade väldiga problem i Ittle Dew, där undantagen är fler än reglerna.

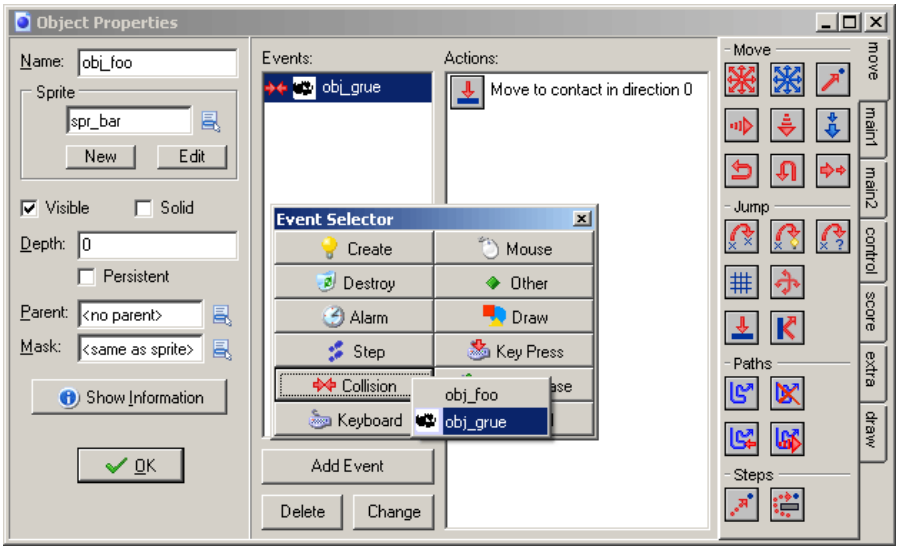

Fig 5. Collision event med ett objekt

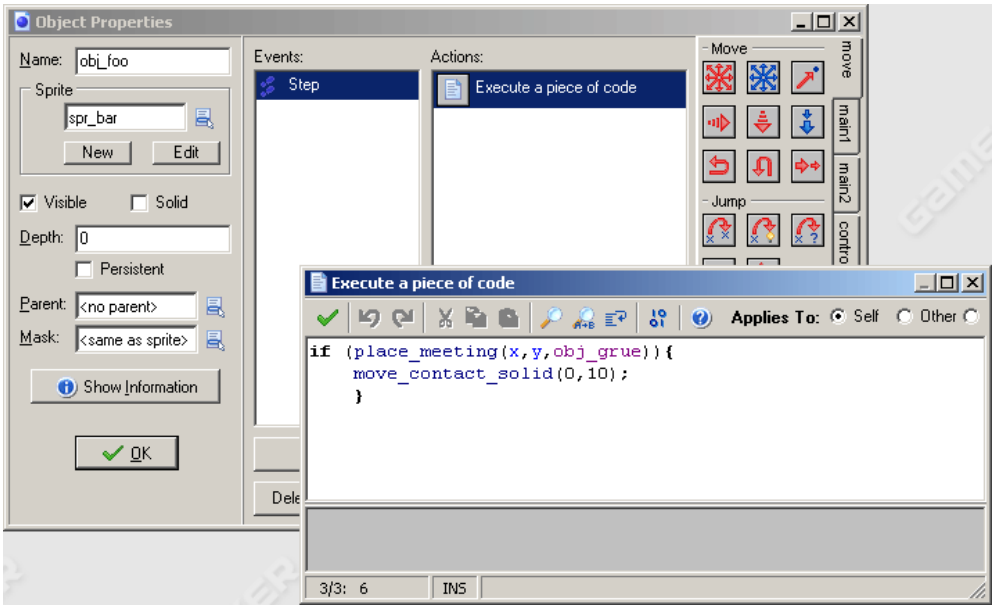

Fig 6. Place_meeting i Step Event. Koden ser till en början ut att göra samma sak som i Fig 5, men skillnaden är att spelet inte vet vilken *instans av obj_grue som kolliderats med, bara att kollisionen skett med någon av dem.*

Det finns ett par sätt att få identifieraren från ett objekt med place_meeting, men de är inte garanterade att fungera. Funktionen *instance_nearest()* returnerar identifieraren på den instans som ligger närmast, och om en kollision med denna objekttyp skedde på samma frame, är det oftast den kolliderande instansen som ligger närmast. Om ett objekt ska kollidera med ett annat objekt som det bara finns högst en instans av, behövs naturligtvis inte denna lösning, utan det unika objektets variabler kan hämtas och

jämföras direkt. Detta kan t.ex. utnyttjas genom att låta fiendeobjektet kontrollera kollision med spelarens avatar istället för tvärtom.

4.3.3 Sparsystemet

Game Maker har en inbyggd ”save state” funktion, normalt konfigurerad till F5 för sparning och F6 för laddning, som gör det möjligt att spara ett spels nuvarande läge till en fil. Detta har dock flera nackdelar:
- Sparfilerna är strikt versionsberoende. En gammal sparfil kan laddas efter att spelet har modifierats, men detta kan innebära allvarliga körtidsproblem.
- När en sparfil laddas, precis som när ett fullskärmsspel tappar fokus, tömmer Game Maker i regel sitt minne av spelet, vilket innebär att grafik kommer att flimra eller bli omringad av svarta rutor, och vissa resurser kanske inte kommer tillbaka alls. På långsamma datorer innebär detta en stor ansträngning för Game Maker och ser inte heller bra ut.
- Sparfilen tenderar att bli stor och ta tid att spara, ju större spelet är.
- Det är enkelt att redigera sparfilen med rätt kunskaper.

Sparsystemets främsta användningsområde är för buggtestning, eftersom det är enkelt att spara en fil, utföra en handling och se vad som händer, för att sedan ladda filen och göra det igen eller annorlunda, så länge spelmakaren inte försöker ladda filer från gamla versioner av spelet. Tillsammans med ”debug mode” är detta ett bra sätt att finna lösningen på oväntade problem under utvecklingen. Om spelmakaren istället vill ha ett snabbt, versionsoberoende och säkert sparsystem bör detta göras manuellt i kod genom att läsa och skriva till textfiler. Dessa kan även enkelt krypteras med checksummor om så önskas.

I Ittle Dew används ett enkelt system med textfiler. Spelaren kan vidröra en ”sparstation” som skriver en fil innehållande spelarens utrustning och position. Denna fil läses in via huvudmenyn, och spelet placerar spelaren i den avsedda positionen med korrekt utrustning.

4.4 Metoder för att snabba upp ett spel

4.4.1 Instansdeaktivering

De kostsamma *instance_activate* funktionerna är ett måste om användaren vill skapa ett stort rum med tusentals till hundratusentals objekt. Med *instance_deactivate* kan alla objekt, särskilda objekt eller alla objekt innanför eller utanför en region deaktiveras. De slutar då att utföra alla sina handlingar, och verkar för spelaren upphöra att existera. Det enda sättet att få tillbaka dem är med *instance_activate*. Normalt vill användaren deaktivera alla objekt utanför spelarens vy, men värt att tänka på är att dessa funktioner inte behöver anropas varje frame, utan bara så ofta som vyn faktiskt rört sig ett visst avstånd. När antalet instanser växer kommer *instance_activate* funktionerna normalt annars att utgöra de mest krävande beräkningarna i spelet.

4.4.2 Reducera antalet kollisionsobjekt

Till att börja med är pixelperfekt kollision[1] mer kostsam än bounding box kollision[2] (detta kan ställas in för varje sprite), men om spelet använder väldigt många block till kollision, exempelvis i ett plattformsspel, hjälper det att skapa större blockobjekt och sätta ut dessa tillsammans med de mindre där de passar (Fig 7). Detta kan kraftigt reducera antalet objekt som rörliga instanser i spelet behöver beräkna sin kollision mot, och reducerar dessutom filstorlek och laddningstid i stora spel.

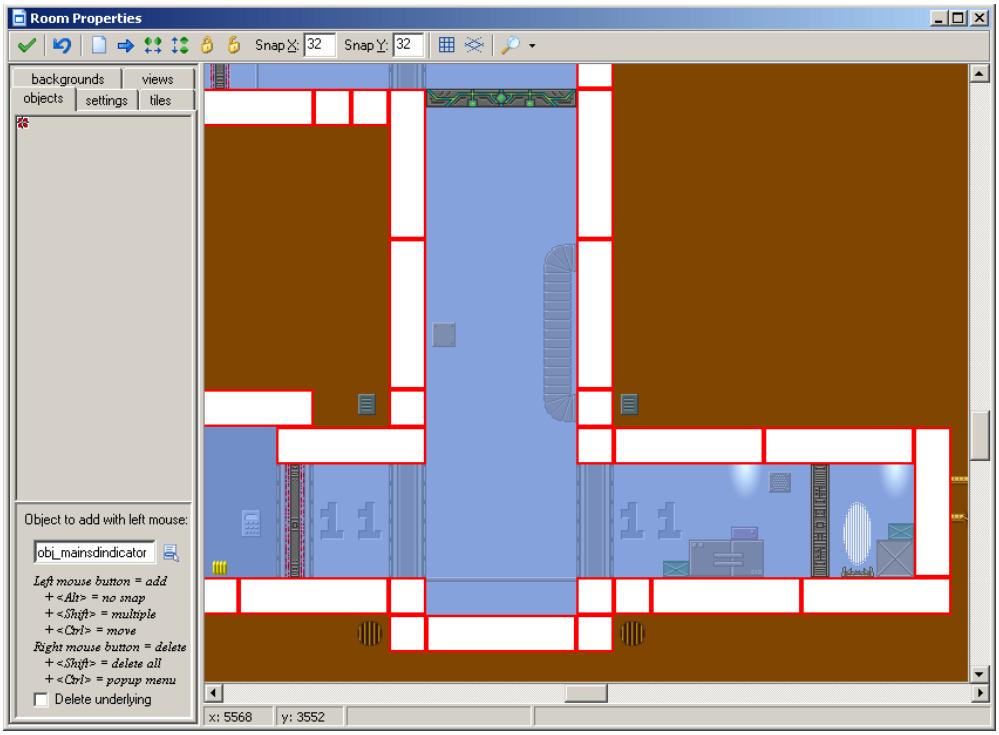

Fig 7. Alla "kollisionsobjekt" I ytterkanten av denna spelvärld har tydliggjorts. Vissa är större än andra vilket gör att det totala antalet kollisionsobjekt har reducerats. Bild från mitt spel Iji (Remar Games 2004-2008)

4.4.3 Reducera antalet tiles

Tiles är bakgrundselement som plockas från en "palett" av många små ytor, för att bygga upp ett bakgrundslager. En yta kan t.ex. se ut som en vägg, en trappa, ett fönster etc. Med andra ord skapar de tillsammans en stor mosaik byggd av dessa små ytor. Det är dock kostsamt att rita ut många bakgrundstiles, varför det är bättre att använda färre, större tiles. Vill användaren minska minnesanvändningen kan även stora ytor av upprepande tiles bytas ut mot större tiles, precis som optimeringen med antalet kollisionsobjekt (Fig 8).

[1] Pixelperfekt kollision innebär att varje pixel på en sprite som inte är genomskinlig räknas. Detta kan användas till exempelvis runda sprites där en fyrkantig kollisionskontroll inte är att föredra.
[2] Bounding box kollision utnyttjar en enda stor rektangel som kollisionsyta.

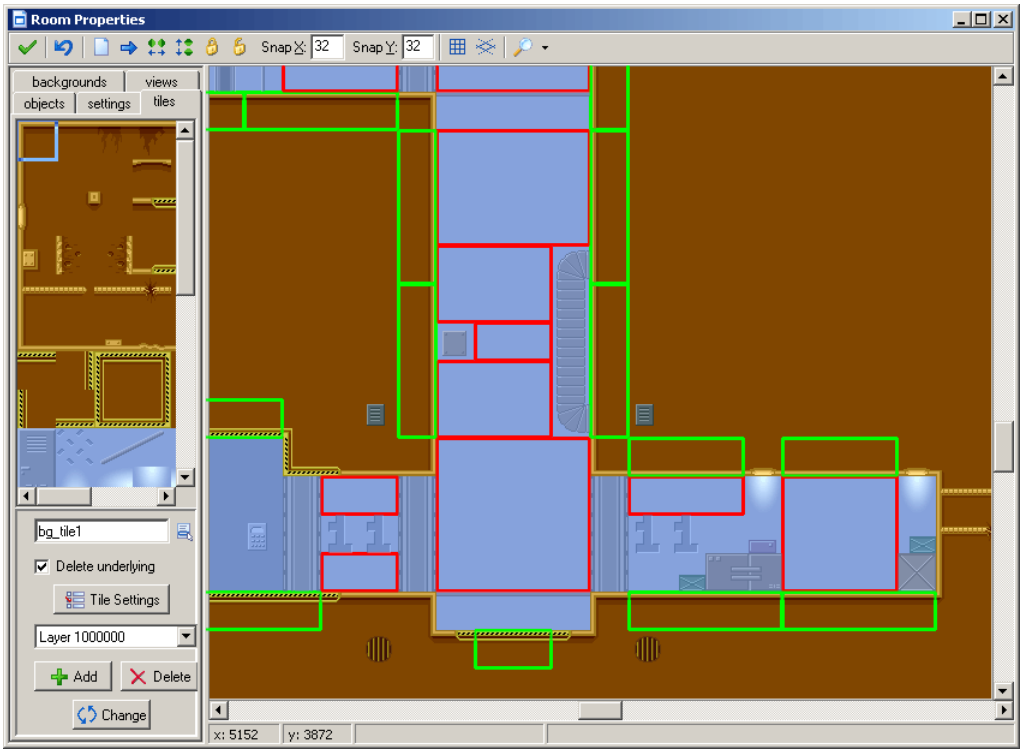

*Fig 8. Alla tiles större än 32*32 pixlar har tydliggjorts. De stora blå ytorna och mycket av väggarna har delats upp för att reducera det totala antalet tiles. Värt att notera är även att det stora bruna området utanför spelvärlden inte är byggs av tiles, utan endast är en generisk bakgrundsfärg. Bild från mitt spel Iji (Remar Games 2004-2008)*

4.4.4 Grafikkompression

Game Maker använder ett komprimeringssystem för sin spelgrafik som gör att bilder med få färger och stora enfärgade ytor tar väldigt lite plats, medan bilder med väldigt många färggradienter och detaljer tar relativt mycket. En fotorealistisk fullskärmsbild kan med andra ord ta flera Megabyte, medan en lika stor helvit bild knappt tar någon plats alls. Dock spelar detta inte särskilt stor roll när spelet väl startas upp, eftersom all grafik "packas upp" till videominnet i alla fall - detta innebär att en fullskärmsbild tar lika mycket kraft och minne i körtid oavsett vad den innehåller. Detta fastställdes genom ett enkelt experiment där jag placerade en fotorealistisk och en helvit bild av samma storlek i varsitt spelprojekt (vilket annars var helt tomt, förutom ett blankt rum för att kunna starta spelet) och kontrollerade filstorlek och minnesbelastning i körtid.

4.4.5 Autogenererade instanser

Detta är inte nödvändigtvis bundet till Game Maker, men på grund av dess ibland begränsade funktioner kan det vara fördelaktigt att dela upp objekt i flera underobjekt. Exempelvis finns det fyra dörrar i Ittle Dew, en för varje riktning, för att underlätta när kartan för rummet ska räknas ut. Det finns dock bara ett ghost-objekt för dörrar, och när spelet konverterar det används dess X och Y koordinat för att räkna ut vilken sorts dörrobjekt som ska placeras ut.

5. Slutdiskussion

5.1 Samarbetet med Joel

Samarbetet gick till en början utmärkt. Vi valde snabbt ut ett spel bland våra koncept och skrev ett enkelt designdokument på det. Dock spenderade vi kanske för lite tid på designdokumentet, eftersom så många totalt oväntade designfrågor och problem dök upp under arbetets gång, och det var onödigt att planera in tid för att rita spelets grafik, eftersom det inte behövde någon färdig grafik för att vara spelbart (i slutändan gjorde jag den heller inte, utan spelet fick nöja sig med temporär grafik). Under implementationen borde jag egentligen ha varit helt avskiljd från Joel, men det hände oftare och oftare att vi hittade på nya saker som inte stod i designdokumentet, t.ex. att det skulle kunna finnas fler än en dörr på samma vägg.

Det rådde också en tvist om ifall spelet skulle kunna fortsätta implementeras och färdigställas efter att arbetet var över. Jag var emot förslaget, varför jag inte spenderade så mycket tid på färdigställning och polering, utan siktade på grundimplementeringen. Detta ledde till at Joel till slut tog över implementationen efter att jag var färdig, för att lägga till saker som start- och slutskärmar.

Det var enkelt att dela upp arbetsuppgifterna och vad respektive person skulle skriva sin reflexiva rapport om. Jag bestämde mig för att skriva om Game Maker, medan Joel skrev om kompakt leveldesign.

5.2 Värdet av prototyping med Game Maker

Istället för att redovisa slutsatser som andra undersökningar inom generell mjukvaruprototyping dragit, kommer jag här endast att diskutera mina egna erfarenheter.

Game Maker har visat sig vara ett snabbt och resursfullt verktyg för att skapa och prototypa enkla 2D spel fort. Allting som stod i designdokumentet för Ittle Dew och lite till kom med i spelet, inte bara som individuella moduler, vilket prototyping också kan vara bra för, utan som en helhet och ett fungerande spel. Visserligen har jag fyra års erfarenhet med programmet, vilket innebär att jag inte har behövt gå igenom någon inlärningsfas under projektets gång - av erfarenhet har jag hört sägas av nybörjare av Game Maker inte verkar kunna uppfylla deras krav, men det verkar snarare ha berott på deras ovana som programmerare. Game Maker är ett program grundat i traditionell programmeringslogik, med objekt, instanser och variabler, och därmed har mycket gemensamt med objektorienterade programmeringsspråk. Därför tror jag att det skulle gå fortare för en programmerare (eller en designer med insikt i programmeringslogik) att sätta sig in i Game Maker till den grad att hon skulle kunna tillverka önskvärt snabba prototyper. Programmet har begränsningar och egenheter, men förhoppningsvis underlättar kapitel 4 för att förstå dem.

Angående värdet av prototyping i sig, anser jag att de många oväntade designfrågorna under projektets gång pekar på att det kan finnas en mängd oplanerade detaljer i ett spels produktion, som utan prototyping inte dyker upp förens det är "för sent" när spelet väl skapas på riktigt. Det är även det enda sättet att säkerställa att det färdiga spelet är roligt att spela. Båda dessa anser jag är de starkaste argumenten för processen. Ittle Dew använde sig även av en mini-prototyp i sig själv; medan vi utvecklade "ghost"-processen testade vi det först med en prototyp, för att vara säkra på att det skulle fungera.

Nackdelen med skapandet av Ittle Dew är att fler detaljer lades på spelet än det var tänkt från början. Det är inte alltid önskvärt att låta skaparen av en prototyp sväva ut och lägga detaljer på detaljer av saker som kommer att vara väldigt krångliga eller kräva alldeles för många specialfall i det riktiga spelet. Det märktes att Joels och min vision av spelet skiljde sig, vilket inte alltid var bra när jag implementerade efter designdokumentet utan att höra vad han hade att säga om det.

Eftersom Ittle Dew egentligen inte var en prototyp av ett riktigt kommersiellt spel, kan mina argument sakna trovärdighet och tyngd. Därför frågade jag en bekant, Justin Leingang som är "Creative Director" på en *Aspyr Media* studio, om hans tankar kring Game Maker, som hans grupp använt för att prototypa sina kommersiella spel. Nedan följer Leingangs svar på mina frågor (personlig kontakt, 5 april, 2008):

> An overview of our prototyping process
> We use Game Maker to prototype every single project that we develop. It's an invaluable tool that allows us to rapidly experiment and execute various components of a game design before ever making a commitment or an assumption that said component will work and/or be engaging. Additionally, this process allows the Game Design team to generate a tangible and playable vision that the rest of the team can soak in, sync up to, and feed directly back in to. As a result, the entire team gains an understanding of what we are all working toward as an end result. It allows us to properly understand and flesh out a realistic scope for the project, and to discover the key components that we should focus most of our efforts on.
>
> Here's a rough chronology of how our prototyping process works:
> - The Lead Game Designer and/or Game Director formulates a basic starting "purpose" that's to be achieved by the core game mechanics.
> - The very basic foundations of a player character package are built in Game Maker. Nothing too complex - just enough to create a framework to tinker within and explore/discover engaging components.
> - Once the most engaging components are zeroed in, and really seem to achieve the purpose, then they are fleshed out further and continually iterated until the player character package is proven.
> - Level components are prototyped according to the features of the player character package - I.e. the various components external to the character that the player interacts with. This includes terrain features, objects, hazards, enemies, etc. This allows us to gain a solid understanding of what types of cause/effect relationships

and interactive situations are most engaging (and also always allows us to discover even more amazing things that we'd have never dreamed of before).
- With the player character package in place, and external components prototyped, a "lesson plan" for the game design is now laid out.
- Level design is planned out according to the proposed lesson plan. Basically, a matrix is planned out that describes the terrain features/challenges and enemy population of each level so that every individual level has its own focus of engagement and player education.
- Each level is now planned via paper maps and illustrations according to the lesson plan.
- The levels are built in Game Maker according to their paper plans, and iterated or completely reworked/scrapped according to the end result.

Min fråga: *Did GM meet your demands?*
Yes, and in fact, Game Maker far exceeded our demands as a prototyping tool.

Min fråga: *More importantly, did you bump into design questions while prototyping that you hadn't thought of when designing the game?*
There's no amount of prototyping in the world that won't introduce questions, iterations, and entirely new concepts into the game design. Prototyping is THE MOST IMPORTANT phase of developing any game (or any interacitve creation, for that matter). It proves and grows a game design. Anybody who calls him/herself a Game Designer and only designs their systems and mechanics on paper is kidding themself and seriously holding back the potential of the end result.

Min fråga: *Or in other words, was the prototyping actually useful?*
If we hadn't prototyped our projects, then they would have either never been completed, or 1000% inferior as game designs. More importantly, the team would have suffered as they likely would have had to rework systems and art on a regular basis - throwing away time, energy, and morale in the process.

Från Leingangs svar drar jag slutsatsen att Game Maker inte bara är ett prototypverktyg för hans grupp, det är en viktig del i deras arbetsprocess. Visserligen verkar han ibland överdriva sina beskrivningar, men av min erfarenhet med honom är det endast hans sätt att skriva. Meningen bakom orden är ändå att GM tjänar sitt syfte som prototypverktyg i detta fall.

5.3 Game Maker som verktyg för färdiga spel

Paul Eres, en medlem i teamet *Radical Poesis Games & Creations*, svarade på min förfrågan om hans erfarenhet i kommersiella Game Maker spel. Detta är hans fria tankar kring ämnet (personlig kontakt, 1 april, 2008):

Hanako Games (aka Hpapillon) was as far as I know the first person to use GM to create commercial games, back during GM5 I believe, and is still by far the most successful at it, and does make a living at it, having sold thousands of copies of

> some of her games (like Cute Knight and Fatal Heart). One of them (Cute Knight) even had a publishing deal from Activision and can be found in stores in some countries and on Amazon.com. Others have created commercial GM games too: The Magi, Shellblast (now freeware?), Sketch Warriors 2, An Untitled Story (now freeware), my game Immortal Defense, and there are probably many others that I don't know about, because it's often difficult to tell what engine a game was made in just by playing the finished product, and most of the people who buy games don't know or care what tool you used to make them.
>
> There are disadvantages and advantages of using GM for commercial games; advantages: easy to use, much of the low-level stuff like collision detection is already coded for you, and there's an active community that will help you figure it out and usually help you playtest your games; disadvantages: it's an interpreted language so performance may be slow in some types of games, it's currently Windows-only, its resolution-change function usually messes up the placement of the player's other windows, the company that owns it is often unresponsive to questions and is in general incompetent, it's only really useful for 2D games.
>
> It's not the most popular engine to create shareware in, but its use is rising among shareware game authors, although they still tend to use mostly BlitzBasic, Blitz3D, Torque, the PopCap framework, or plain languages like C++, mainly because most of those are cross-platform and the Macintosh platform is very lucrative for shareware, with a higher average conversion rate (defined as sales per demo download) than the PC platform has. Also, GM6 had conflicts with a few popular digital rights management wrappers, although this may no longer be a problem for GM7.

Jag tolkar Eres svar hittills som att det finns en marknad för spel gjorda i Game Maker, lika väl som det finns en marknad för spel skrivna i andra språk - med den viktiga skillnaden att Game Maker spel endast är kompatibla med Microsoft Windows-plattformen. Den andra största svårigheten verkar ligga i att nå ut med sin marknadsföring som självständig utvecklare. Det hjälpte troligtvis mycket för Hpapillon att skriva ett avtal med distributören Activision; själv minns jag endast att jag sett hennes spel upplagda på vad som på den tiden var Game Makers officiella Internetforum, vilket hade en relativt minimal användarbas jämfört med Amazon (www.amazon.com).

> And here's a short history of my own game: I began Immortal Defense in early December 2006 and released it on June 1, 2007. Unlike most shareware games, I shared the authorship of the game among several people, and each person gets a percent of the profits depending on how much work they did for the game, whereas in most shareware games one person either does everything or pays others to do other parts, offering no royalties. I chose my way because I worked with friends and because I feel it's an incentive to do a good job if you have a stake in how well the game sells.

> I had been playing a particular online Flash game in the tower defense genre and got so into it that I wanted to make one of my own, but I wanted to make it better and feel more like a game, so I added a focus on story, I had a 100-level system where your scores in earlier levels affected your resources in latter levels, and I let the mouse cursor itself be more involved in the action by letting it fire and use special attacks. Most tower defense games at the time were free (with the exception of the shareware game Master of Defense), so it may not have been the best genre to start out with, but it was one I was enjoying at the time and one I felt had a lot of room for growth.
>
> I got the basic game working in the first three weeks or so; the game actually was completely playable to the end at that point, it just lacked very much polish and didn't have most of its resources (graphics, sound, music, story text) in yet. The next few months were spent polishing and tweaking everything, finding and removing bugs, and getting all the resources in. There were no major bugs except one that took several weeks to solve which didn't happen on my computer but crashed other people's computers, which was pretty difficult to solve because I had to get people to try my changes over and over, repeatedly crashing their computers just for my sake. I eventually narrowed it down to an issue with drawing primitive triangles where if you don't draw the final point in a triangle but just draw two points, some video cards don't know what to do and others can handle it without issues. Fixing this bug actually took more time than the development of the core of the game itself.
>
> After release it sold better than I expected it would, it averages about 25 sales a month, with good and bad months. Because it was my first commercial game, I had little experience in marketing so for a while I tried a different tactic each week or so to market it; one week was forum announcements, another week was submission to demo hosting sites, another week using Google ads, and so on, learning what works. It got very positive reviews and fan mail, and Game Tunnel's Independent Strategy Game of the Year; sometimes I think the reviewers like it a lot more than I do -- not that I dislike it, but sometimes their praise for it is far above anything I'd ever give to any game, by anyone. Generally the best-received part of the game was the story and in particular its ending, perhaps because most shareware games don't have stories at all or have fairly generic ones, whereas I had a professional writer friend of mine write it.
>
> The game's history isn't over yet because I'm still working on an update with a level editor and other improvements, and I'm still planning to get the game on "portals" like Manifesto Games and Reflexive, which tend to have a lot more traffic than only selling it off an individual game developer's site.

Eres gjorde sitt bästa för att nå ut och sälja spelet på olika sätt. Han verkar nöjd med resultatet, men uppenbarligen skall utvecklaren av ett spel som Immortal Defense inte förvänta sig att sälja spelet i tusental, när det saknar PR-maskinerna som de stora kommersiella spelen, uppbackade av stora företag med väldigt mycket pengar, har.

Det finns en sista svaghet med Game Maker som bör tas upp, vilken jag vidrörde i kapitel 4.1.3 men Eres inte nämnde: det är möjligt med rätt program att utföra en "decompile" på ett färdigt spel, och erhålla den redigerbara filen som spelutvecklaren jobbat med. Detta betyder att vem som helst i teorin kan redigera och byta ut vad som helst i spelet, antingen för att själv ta åt sig äran för att ha skapat det, för att kunna sprida det gratis, eller för att smutskasta skaparen. Game Maker "decompilers" är olagliga, men lätta att hitta på Internet, och endast en version av programmet kan i skrivande stund stå emot ett decompiler-program, nämligen ett GM6 spel som konverterats till GM7 med verktyget som nämndes i kapitel 4.3.2. YoYoGames har i skrivande stund valt att ignorera problemet istället för att ta upp det för diskussion, vilket i min mening endast förvärrar situationen. I praktiken har det i min erfarenhet ytterst sällan hänt att någon redigerat ett spel och försökt ta åt sig äran, men trots osannolikheten att det händer - särskilt om spelet utannonseras långt i förväg så att det inte råder någon tvekan om vem skaparen är eller vad hennes intentioner med spelet är - kan det vara något att tänka på.

Till sist, angående färdiga spel som inte är kommersiella; mina egna spel No Friction (2004), Hero (2005), Retrobattle (2005), Castle of Elite (2006) och Garden Gnome Carnage (2007) har laddats ned i sammanlagt över 25000 exemplar från min hemsida (och ett oräkneligt antal från andra sidor som "speglat" spelen) och det finns ännu mer framgångsrika personer, som "Pug Fugly", "YoMamasMama", "BitPimp" "Srehpog" och den framlidne "Shawn64" inom enmans Game Maker produktioner. Det är fullt möjligt att skapa roliga, spännande och långlivade spel, så länge användaren inte tolkar namnet "Game Maker" som att spelet kommer att programmera sig självt.

Referenser

Ousterhout, J.K. (1998) Scripting: Higher Level Programming
for the 21st Century. *IEEE Computer* magazine, *March 1998* Tillgänglig på Internet: http://home.pacbell.net/ouster/scripting.html (Hämtat den 8/4 2008)

Schildt, H. (2000) *C: The Complete Reference, Fourth Edition*. Osborne/McGraw-Hill

Smith, M. F. (1991) *Software Prototyping: adoption, practice and management*. McGraw-Hill

Mark Overmars (199?-2004) *Game Maker* (Version: 5.3A). [Datorprogram] Tillgängligt på internet: http://www.gamemaker.nl/old.html

Mark Overmars (200?-2007) *Game Maker* (Version: 6.1). [Datorprogram] Tillgängligt på internet: http://www.gamemaker.nl/old.html

Game Maker (Version: 7) (2007-2008) [Datorprogram] YoYo Games. Tillgängligt på internet: http://www.yoyogames.com

The Legend of Zelda (1987) [TV-spel] Nintendo Co., Ltd.

Bilaga A: Designdokument för Ittle Dew

Avataren
Gå
Putta
Tryck mot object viss tid för att putta det. Avatar står kvar (om inte fortsätter att trycka). Alla block, isblock, bomber etc som kan knuffas snappar till tiles.

Verktyg:
- Portal Gun
- Ice wand
- Bomb
- Melee
  - Stick
  - Fire sword
  - Flame Thrower

Sköld
- Framme i normalläge
- Försvinner när man använder verktyg
- Uppgraderas i steg
  - Normal
  - Big – skyddar mot större projektiler
  - Mirror – Reflekterar projektiler

Kontroller
Styrs med joypad. D-pad + 4 face buttons
Varje verktyg har egen knapp.

Verktygen
Portal Gun
Tryck 1 gång för att skjuta ut destinations-boll. Relativt låg hastighet.
Trycka 2'a gång för att skjuta ut teleport-boll. Denna har högra hastighet. Det objekt den träffar teleporteras till destinationsbollens nuvarande position. Om teleportbollen inte träffar ett relevant objekt så sker ingen teleport. Destinationen försvinner när teleportbollen försvinner.
Båda bollar studsar mot reflekterande ytor.

Uppgraderingar:
- Första versionen kan inte teleportera knuffblock/isblock
- Andra versionen kan teleportera även knuff-och isblock

Ice Wand
Skjuter ut isboll. Isbollen formas en eller två tiles framför spelaren.
Isboll fryser monster, spelaren, block, bomber. Frysta objekt glider extra långt när de puttas. Frysta objekt töar efter viss tid.
Isboll släcker eldar. Detta kan utlösa en trigger.
Isboll studsar mot reflekterande ytor.

Om en titan (monster som blir block när det fryses) försöker frysas men hamnar på samma tile som avataren, fryses det inte utan isbollen försvinner istället.

Bomb

Oändligt förråd. Bara en kan finnas ute i taget. Bomber läggs ut en tile framför spelaren.
Frysta bomber har avpausad detonering. Vid upptining fortsätter detoneringstiden.
Bomber kan puttas en tile i taget. Frysta bomber fungerar som alla frysta block och glider.

Meleevapen

Stick (pinne)

Slår framåt, kort räckvidd.
Kan slå fiender, triggar triggers.
Man kan sätta eld på pinnen så får den eldattribut. Slocknar efter ett tag.

Fire sword

Slår framåt, lång räckvidd.
Eldattribut. Smälter is och tänder bomber.
Starkare än pinnen.

Flamethrower

Skjuter eld framåt, en salva i taget. Elden försvinner när den träffar väggar och triggers.
Mycket lång räckvidd. Når över hål.
En ”bit” av elden kan teleporteras om den träffar en teleportboll.
Starkare än eldsvärdet.
Sätter eld på féer. Röjer fett.

Level objekt

Triggers

Kan:

Öppna en låst dörr
Trolla fram en pinata

Pressure pads – kan triggas av block, frysta stora fiender, bomber.
Kristall – triggas av slag med melee, bombexplosion, ice wand, dest/teleportboll.
Door switch – sitter precis jämte dörren, annars som kristall
Eldar – Släck med ice wand för att trigga
Släck eld – tänd med eld för att trigga
Triggers som är aktiverade förblir aktiverade efter de ”använts” (den öppnade dörren har passerats, den framkallade piñatan är öppnad).

Monster no zone

En tile. Monster går inte hit. Används för att blockera monster.

Block no zone

Ittle no zone

Block kan färdas över dessa

Reflekterande ytor

Reflekterar alla typer av projektiler. utgångsvinkel inverterad mot ingångsvinkel.

Speglar, kan luta 45 grader.
Isblock är reflekterande.
Mirror shield är reflekterande.
Is-vägg – bildas av att skjuta ice wand in i väggen. Is-vägg töar efter ett tag, och förstörs omedelbart av bombexplosioner och eld (väggen blir en vanlig vägg igen).

Block

Kan puttas en tile i taget, endast om det inte finns något i vägen på nästa tile, inklusive fiender. Puttning sker genom att avataren trycker sig mot blocket i en viss tid.
Triggar pressure pads så länge det är direkt över dom.
Block kan frysas med Ice wand och blir då isblock som glider tills det träffar en vägg när det puttas. Töar efter viss tid. Töar även när de träffas av bombexplo / eld.

Isblock

Kan puttas – glider tills det träffar en vägg.
Triggar pressure pads sålänge det är direkt över den.
Töar aldrig på egen hand.
Förstörs av bomber och eld.
Är en reflektor.

Piñatas

Innehåller skatter så som nya verktyg, hälsouppgradering etc.
Öppnas genom att slå med melee
Finns max 1 per rum.
Med hjälp av en trigger kan piñatan göras oanvändbar (den sitter bakom galler eller liknande). Den görs användbar igen med en (elelr flera) triggers, dvs aktiveras precis som en dörr.

Dörr

Kan öppnas med triggers av olika slag.
En upplåst dörr förblir upplåst. (När alla tillhörandetriggers är aktiva samtidigt låses dörren upp permanent)

Sjunkande pelare

Kan stå mitt i rummet.
Blockerar vägen.
Sjunker ner i golvet när triggad
Kan inte interageras med i övrigt.

Trappa

Placeras i rummet.
Leder mellan plan.

Old Man Kundvagn / Father Forah

Finns utspridda över hela borgen. Säger kryptiska ledtrådar (en per gubbe) som har att göra med en närliggande hemlighet / pussel.
En special-gubbe finns i ett speciellt rum. Hasplar ur sig fånigheter och halvsanningar på slump när man går in i rummet. Har en stor mängd repliker att slumpa mellan
Exempel: ”Use buttons to win the game.”

Fiender

1. Faerly Harmless

Undviker pinnen, flame sword, bomber, ice wand, portal gun, men kan träffas.
Kan grillas med flamethrower.
”Hey!” ”Watch out!” ”Hello!”
Ingen attack.
Rörelsemönster: Flyger runt och är irriterande. Spelar egentligen ingen roll hur den rör sig.

2. Jenny Rich Monsteur

Vanlig mob.
Finns i tre färger: grön, gul, blå.
Grön: Går runt och petar med svaga attacker. Kan blockas med skölden.
Gul: Kastar yxor. Kräver big shield eller bättre.
Blå: Skjuter med laserpistol. Kräver mirror shield. Farliga.
Fryser man den stannar den.
Rörelsemönster: Går mest runt på slump och attackerar när spelaren kommer i skottlinje. Ju lägre rank, desto mer slump, och ju högre rank desto mer jagar den aktivt spelaren.

3. Titan Appde’Grafiques

Tung, hård, fyrkantig form. Blir isblock när de fryses.
Melee attack, kan stoppas med big shield eller bättre.
Rörelsemönster: Hoppar en tile i taget. Rör sig slumpmässigt tills spelaren kommer tillräckligt nära, då blir hon jagad. När den är i luften kan den inte frysas.

4. Noselol / Stack of pancakes

Äter bomber och portaler. Bomberna/portalerna försvinner, fienden lever. Jagar närliggande bomber och portaler.
Fryser man den stannar den.
Fångar (äter) spelaren, man måste slå sig ut så dör den. Ingen sköld hjälper mot att bli uppäten.

5. Turnip

Svänger höger när den träffar en solid vägg eller block. Den börjar alltid med att gå uppåt. Puttar bomber och block framåt en tile och svänger då.
Fryser man den stannar den.
Elektrifierar spelaren om hon kommer för nära (area attack). En tiles räckvidd. Ingn sköld hjälper eftersom det är en area attack.

6. Ismonster

Keldas Darknuts. Har reflekterande skydd framåt, massiv sköld.
Svåra att döda. Eld och bomber skadar dom mer. Får tillbaka hälsa av isattacker.
Attackerar med ice breath. Når två tiles. Fryser Kelda, man får masha sig ut. Ice breath stoppas med big shield eller bättre.

7. Eldmonster

Man kan tända pinnen på den, men monstret tänder inte eld på saker själv.
Alla meleevapen är ineffektiva. All eld är alltså ineffektiv.
Försvinner när man skjuter den med ice wand.
Bomber blåser ut dom.
Saknar attack, men Kelda skadas av att nudda dom. Kelda får eld i baken och springer runt svårkontrollerad.

Hälsa
Börjar med 1 baguett.
1 baguett kan skadas i 4 steg.
Genom att hitta 4 baguettdelar i piñatas så bildar man en hel ny baguette.
De fyra delarna är grafiskt annorlunda från varann, men man hittar alltid nästa del i turordningen.
- Bröd
- Ost
- Sallad
- Tomat/Pastrami/Jordnötssmör/Kolasås – beroende på vilken baguette man samlar till

Man kan samla ihop till totalt 4 baguetter.

Gränssnitt
HUD visar baguetter och de 4 verktygen, samt minimap

Pausemeny
- Visar alla upgrades
- Full karta
- Load checkpoint
- Exit Game

Startmeny
- Start
  - 1, 2, 3
- Load
  - 1, 2, 3
- Options
  - Sound level
  - Music level
  - Set buttons...
    - Bla
- Exit

Karta
Minikartan fylls i efterhand man kommit in i nya rum och hittat skattkistor.
- Alla rum är gråa tills man varit i dem.
- Beträdda rum blir vita, och rum-kopplingar och dörrar visas
- När man hittat kartan blir obefintliga rum helt svarta, och obeträdda förblir gråa

Kan hitta kartor över hela planet som visar:
- Dörrar (om dom ännu inte är permanent öppnade)
- Skattkistor
- Bombade hål
- Verktyg visas med extra viktig ikon.

Finns bara en piñata eller trappa per rum, den visas mitt i rummet.
Kartan i pausemenyn visar alla våningar i mer detalj.

Sparning
Finns utplacerade spar-stationer som måste aktiveras genom att slå på med melee.

När man dör får man valet att spara, avsluta eller fortsätta. Om man väljer fortsätt så sparas spelet samt man hamnar på den senast aktiverade sparstationen.
När man spawnar så har man kvar alla items + uppgraderingar, men har lika mycket liv som man började med.
Finns 3 sparprofiler

Boss

Vi får se vilken som hinner göras:

1
Lite grafik och lite dialog, men ingen bossfight. Lal!

2
Lite grafik och dialog, man kämpar mot bossen genom val i dialogen.

3
Lite grafik och dialog. Man får spela ett minispel mot bossen, typ Pong.

4
Full-fjädrad boss utan fjädrar. Man måste använda items etc. Bossen har kanske flera steg.

Endgame
Vi får se vilken som hinner implementeras:

1
Cutscene bestående endast av stillbilder och text.

2
Cutscene bestående lite av in-game med bossen, och sedan med stillbilder och text. En resultatskärm dår det står tid, pengar, kanske lite stats.