Mjukvaruarkitektur/Teknisk Dokumentation - Spel

# Teknisk Dokumentation

## Vilka program används?

För att utveckla spelet använder vi oss av spelmotorn Unity (5). För programmering använder vi Visual Studio 2012 och Monodevelop. Det krävs en del förarbete och konfigurering för att arbeta i Visual Studio, eller annan IDE än Monodevelop då programmen måste anpassas för att fungera med Unity.

För att följa vår valda kodstandard så rekommenderas Visual Studio, då Monodevelop autoformaterar kod på ett sätt som ser annorlunda ut gentemot sättet vi skriver på i Visual Studio.

För att versionshantera vårt projekt använder vi oss av GitHub och deras windows-applikation.

Det kodspråk vi valt till projektet är C#, eftersom detta stöds utav både Visual Studio och Monodevelop. I Monodevelop kan man även utveckla med hjälp utav JavaScript, vilket vi valde bort eftersom projektet då skulle behöva översättas.

Till den grafiska delen har vi använt oss av PhotoShop för att skapa sprites, och Unity’s animationskomponenter för att sätta samman sprites och animera dem.

* Unity 5 Spelmotor (<http://unity3d.com/>)
* Visual Studio 2012 Ultimate (<https://www.visualstudio.com/en-us/products/visual-studio-ultimate-with-msdn-vs.aspx>) (eller annan version) utvecklingsmiljö
* alt Monodevelop (<http://www.monodevelop.com/>) som följer med Unity’s utvecklingsmiljö
* Github (<https://windows.github.com/>) versionshantering
* Adobe Photoshop CC 2014 (<http://www.adobe.com/se/products/photoshop.html>)

## Versionshantering

För versionshantering har vi använt oss utav GitHub och särskilda arbetssätt.

1. Innan man börjar dagens arbete så ska man göra en sync via sin GitHub-klient.
2. Efter varje avslutat del i arbetet / större förändring så gör man en commit med en förklarande titel och beskrivning av vad man gjort.
3. Efter avslutat arbete eller vid behov så pushas alla dagens commits upp via GitHub-klienten.

För att undvika onödiga filkonflikter så har vi i Unity aktiverat så att metadata filerna är synliga, samt lagt till en .gitignore lista med filer som inte ska versionshanteras. Ignorelistan och instruktioner till hur man lägger till den i sitt repositorie kan hittas här:

* <https://github.com/github/gitignore/blob/master/Unity.gitignore>
* <https://help.github.com/articles/ignoring-files/>

## Komma igång med Unity

Ett bra sätt att komma igång och lära sig Unity är att kolla igen några av de många olika tutorials som finns. Vi började med att kolla på en space shooter tutorial som ger en bra inblick i hur Unity fungerar:

<http://unity3d.com/learn/tutorials/projects/space-shooter>

Fler tutorials kan hittas här: <http://unity3d.com/learn/tutorials/modules>

Förutom tutorials har Unity bra dokumentation på alla sina delar som finns här:

Manual : <http://docs.unity3d.com/Manual/index.html>

Scripting : <http://docs.unity3d.com/ScriptReference/index.html>

## Kodstandard

För att göra det enkelt att hitta bland alla våra scripts så har vi lagt fokus på att skapa en bra mappstruktur. Exempelvis hittar man alla scripts som rör fiender i mappen EnemyScripts och spelarrelaterade scripts i mappen PlayerScripts.

Nedan följer exempel på hur vi valt att formatera vår kod.

If-sats med en rad kod

if (lifeText == null)

lifeText = GameObject.Find("HpBarText").GetComponent<Text>();

If-sats med flera rader kod

if (target != null && isInRange)

{

canPatrol = false;

startPatrolTimer = 1f;

FollowTarget ();

if (platformSticky && EdgeCheck ())

{

StopMovement ();

}

}

## Skapa grafik i Photoshop

1. För att skapa karaktärer så ritas varje kroppsdel i ett separat lager.
2. Dessa delar skapas genom att göra former (cirklar, rektanglar etc.) som sedan omformas till kroppsdelar.
3. Delarna färgas och skuggas var för sig.
4. Den fullständiga bilden trimmas och sparas i PS och PNG format.
5. De separata delarna kopieras till nya filer, trimmas och sparas i PS och PNG format.
6. De separata filerna läggs sedan till i Sprite-katalogen i Unity, under lämpligt namngiven underkatalog.

## Skapa animationer i Unity

För en grundläggande genomgång i hur Unity’s animeringssystem fungerar se här:

<http://unity3d.com/learn/tutorials/modules/beginner/animation>

Här beskrivs översiktligt hur vi skapat animationer i Unity:

1. Ett spelobjekt för det tilltänkta tillägget skapas i Unity.
2. Varje del från spelobjektet (finns i spritekatalogen) läggs till som ett child-element och placeras sedan in på tilltänkt plats.
3. Spelobjektet i hierarkin och man klickar på “Create new animation” i animationsfönstret.
4. Animationstiden bestäms och man positionerar om objektets delar under de olika steg som finns under animationstiden.
5. När animationen flyter på som den ska så sparas den.
6. Användaren går sedan över till Animator fönstret för objektet och skapar de parametrar som behövs för att göra animationsövergångar (ex. en bool som berättar ifall objektet rör sig, är på marken etc.)
7. Kopplingar mellan de olika animaitonerna skapas och parametrar sätts på de olika kopplingarna.

## 

## 

## Kopplingen till community

### Inledning

För att kunna få en koppling mellan spelet och community-sidan har vi tagit hjälp av Unity:s hjälpklasser WWW (för GET) och WWWForm (för POST).

<http://docs.unity3d.com/ScriptReference/WWW.html>

<http://docs.unity3d.com/ScriptReference/WWWForm.html>

Datan som hämtas och postas är spelarens lifetime stats (kills, deaths, jumps och guld) och status på achievements.

För information om vad som ska skickas med till servern kolla dokumentet “Instruktion API” i mappen “Teknisk dokumentation”.

### Logga in

En simulering av ett login till community-sidan har gjorts för att förenkla en inloggningsprocess. På servern finns ett script som vill ha användarnamn och ett krypterat lösenord. Dessa jämförs med databasen och skickar tillbaka en JSON-sträng med svaret 0 eller 1 för att berätta om det fanns eller inte.

En GET-request ska skickas till servern med användarnamn och lösenord i headern.

När vi får tillbaka en 1 så har vi en statisk klass CommunityUser som vi sätter som inloggad (isLoggedIn) och sätter användarnamn och lösenord då endast en ska kunna vara inloggad. Detta hjälper till när man ska posta och hämta data från servern. Man kollar alltid isLoggedIn om man ska kunna skicka och posta data. Om den är sann så hämtar man det användarnamn som ska uppdateras.

#### URL

Login: <http://www.metalgenre.se/api/achievements/GetUser.php>

#### Script

*Login.cs*

*CommunityUser.cs*

### 

### 

### Hämta data från community

För att hämta data använder vi WWW för att göra en GET-request till servern. Om man ska kunna hämta kollas först i det script som kallar på hämtning om det finns en användare inloggad i spelet med hjälp av CommunityUser-scriptet.

#### URL

Achievements : <http://www.metalgenre.se/api/achievements/GetAchievement.php>

Stats : <http://www.metalgenre.se/api/stats/GetStats.php>

#### Script

*WWWGetPlayerData.cs*

*GetAchievements.cs*

### Posta data till community

När vi ska posta datat till community använder vi hjälpklassen WWWForm för att strukturera ett formulär med fält som php-scriptet förväntar sig på server. Om man ska kunna posta kollas först i det script som kallar på postning om det finns en användare inloggad i spelet med hjälp av CommunityUser-scriptet.

#### URL

Achievements: <http://www.metalgenre.se/api/achievements/PostAchievement.php>

Stats : <http://www.metalgenre.se/api/stats/PostStats.php>

#### Script

*WWWPostPlayerData.cs*

*WWWPostAchievementData.cs*

# 

# Mjukvaruarkitektur

## Arkitekturellt Signifikanta Krav

Vi valde kraven:

* F1 Kontroller
* F2 Kamera
* F3.1 Workerbot
* F7.3 Liv
* F7.4 Skada
* F9.4 Huset
* F9.5 Karta för val av nivå
* F9.1 Gruvan
* K1.1 Spelare - The Miner
* K2.1 Rörelseanimationer
* K2.2 Attackanimationer

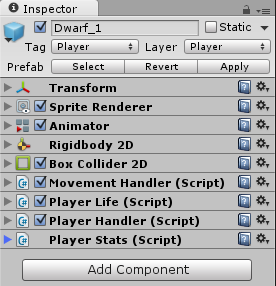
som våra arkitekturellt signifikanta krav.

### Motivation

När dessa krav blivit implementerade kan man säga att vi i vårt fall har ett minimalt spelbart spel. Huset är basen för spelaren och där väljer man nivåer att spela, i det här fallet gruvan. En spelarkaraktär behövs för att göra kontroller och kamerainställning då den ska följa spelaren. Kombinera fienden med spelarens liv och skada så får man ett så kallat “fail state” som ofta ses som en nödvändig feature i en mjukvara för att få klassas som ett spel (ingen direkt standard, något jag hört, Kevin). Alltså att fienden kan besegra dig och du måste börja om igen.

## Struktur

### Hur man får funktionalitet

Vi har baserat vår struktur på hur Unity redan används. Hur man programmerar är att man skapar olika objekt och placerar komponenter på dem. Tack vare Unity får man mycket färdigt (fysikmotor, kollision, användbara funktioner etc) och det vi gör är att sätta så kallade scripts på objekt. Dessa script kallas “behaviour scripts”, alltså att man lägger beteenden på objekt.

Ett exempel kan vara att vi vill att spelaren ska kunna röra på sig med hjälp av tangentbordet. Det man gör är då att på ditt spelare-objekt lägger till en ny komponent (det finns olika sätt att göra detta på) på objektet, väljer att skapa ett script som komponent. Efter det börjar du redigera scriptet i din valda editor (monodevelop, visual studio). Här bestämmer du för att hämta spelare-objektets positon, lyssna på knapptryck och sedan manipulera position beroende på knapp.

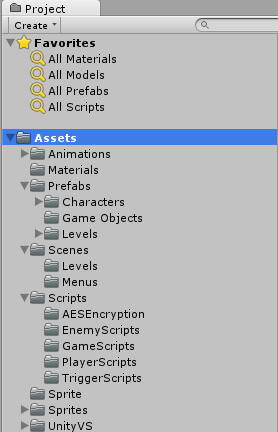
### Hur vi valt att strukturera

#### Logiskt

##### Scenerscene hierarchy.png

I Unity arbetar man i så kallade scener. En scen kan vara ett helt spel eller delar av ett spel. I scenen har man en hierarki med spelobjekt. Ett spelobjekt kan vara vad som helst (spelare, textur, en trigger, ett ui element etc). I vårt spel har vi bestämt oss för att en scen ska vara ett rum på en nivå, så vi delar alltså upp nivåer i rum. Varje scen blir då specialutformad med tanke på rummets layout och objekt osv. För att underlätta läsbarhet och spårbarhet kan man skapa tomma objekt som “mappar” för olika typer av spelobjekt som i bilden.

##### Prefabs

I Unity använder man sig också av så kallade prefabs. En prefab gör man om man vill kunna återanvända ett objekt och dess egenskaper utan att behöva kopiera och ändra individuella objekt i scener. När man ändrar en prefab återspeglas ändringen i alla instanser av den prefaben (kan också override:as på individuella objekt). Prefabs gör vi av nästan allt som vi vill kunna köra flera instanser av, eller för att underlätta när man vill lägga till samma saker i olika scener, några exempel är fiender och användargränsnitt.

##### Script

När vi skriver scripts för våra spelobjekt ser vi till att det som finns i scripten bara har att göra med det objekt vi sätter den på. Men om något annat script behöver kännedom om det scriptet (spelaren behöver tillgång till stats) så ser vi till att göra variabler och funktioner tillgängliga.

#### Filstruktur

Filstrukturen som vi har valt är att helt enkelt lägga script som har med spelaren att göra, i mappen PlayerScripts. Det samma gäller med andra typer, fiende scripts ligger i EnemyScripts. Mer allmäna scripts som hanterar saker på flera olika plan, ligger i GameScripts, osv. Till höger så kan du se på bilden hur vår struktur ser ut.

## Konsekvenser

### Fördelar med strukturen

Det är ganska lätt att få igång något körbart, Unity är en mycket färdig och stabil motor. Man får många byggblock gratis och det vi behöver tänka på är hur spelet ska fungera för att sedan bygga ihop det med blocken.

### Nackdelar med strukturen

Det kan ta ett tag att vänja sig vid hur man programmerar i och använder Unity på ett bra och effektivt sätt. Det är lätt att skapa små enkla spel i Unity men när man ska göra lite större blir det genast mycket mer att tänka på.