Technical Design Document

CIRCUIT

Teammitglieder:

Ben Kantwerk (Team lead)
Andrei Stein
Grigori Jordan
Philipp Hundelshausen

Inhaltsverzeichnis

TECHNOLOGISCHE VORARBEITEN

Code-Konventionen

Variablen

Membervariablen

- privat definieren und Getter/Setter setzen (siehe Getter/Setter)
- vorangehendes m, wenn mehrere Wörter zusammenhängend mit jeweils dem ersten Buchstaben groß
- Member nicht im Klassenkopf bei Definition belegen sondern immer im Konstruktor
  Beispiel: int mExampleVariable

Temporäre Variablen

- d.h. Variablen die nur innerhalb einer Funktion oder Konstruktion (if, while etc) bestehen
- wie Member nur vorangehendes t

Konstanten/statische Member die niemals geändert werden

- Komplett in Großbuchstaben, Wörter durch _ getrennt
- Wertzuweisung direkt in der Definition und nur dort
  Beispiel: int EXAMPLE_VARIABLE = 0;

Spezielle Variablen

- Bestimmte Klasseninstanzen die durchgehend gesondert benannt werden:

SB;	ContentManager CM;
Camera CAM;	GraphicsDeviceManager GD;
GameTime GT;	CollisionSystem CS;

- eventuell weitere z.B. Skybox SKY
- Desweiteren bestimmte Member großer Instanzen wie des ActionScreen
  z.B. List<Enemy> ENEMIES o.Ä.

Zugriffsvariablen in foreach() Konstrukten

- einfach der erste Buchstabe der Klasse auf die zugegriffen wird mit führendem _
  Beispiel: foreach(Enemie _e in ENEMIES)
- bei verschachtelten foreach Konstrukten sind die Zugriffsvariablen zu nummerieren
  Beispiel: foreach(Enemie _e0 in ENEMIES)
  foreach(Weapon _w1 in _e0.mWeapons)

Parameter

- führender _ bei mehreren Wörtern
- Anfangsbuchstaben groß außer beim ersten Wort
  Beispiel: int _exampleParameter
- Spezielle Variablen sind hiervon ausgenommen, sie werden übergeben wie sie sind

Zählvariablen in for() Konstrukten

- einfach i
- bei verschachtelten Konstrukten x, y, z

Funktionen

Allgemeines

- Funktionsnamen werden groß geschrieben
- mehrere Wörter zusammen mit den Anfangsbuchstaben groß

Getter/Setter

- Name identisch zur zugehörigen Membervariable ohne das führende m

Form:

```
public Vector3 Position
{
        get { return mPosition; }
        set { mPosition = value; }
}
```

Überladen

- Nach Möglichkeit die Parameter die allen Überladungen gemeinsam sind zuerst auflisten
- Überladungen direkt hintereinander schreiben d.h. nicht durch Absätze trennen

Beispiel:

```
private void ExampleFunction(int _a, string _b)
{ }
private void ExampleFunction(int _a, float _c)
{ }
```

Events

Allgemeines

- Benennung mit führendem e
- Benennung der beim Feuern auszuführenden Funktion mit führendem On
- Parameter der Funktion sind immer (object sender, EventArgs e) wobei EventArgs ggf durch die jeweiligen Custom-EventArgs ersetzt wird

EventArgs

- Klassenbenennung endet auf EventArgs
- Variablen und der Konstruktor haben die Zugriffbeschränkung internal sofern sie nach außen sichtbar sein sollen
- nach außen sichtbare Variablen werden hier groß geschrieben (keine Getter/Setter)

Zur Veranschaulichung dieser Konventionen wurde außerdem eine Template-Klasse angefertigt.

Weitere Anmerkungen

Da es sich bei diesem Softwareprojekt für alle Teammitglieder das erste groß angelegte 3D-Spiel handelt, ist es besonders wichtig, dass sich die Teammitglieder in ihren speziellen Aufgabenbereich einarbeiten müssen, um den hohen Anforderungen dieses Projektes gerecht zu werden. Besonderes Augenmerk liegt dabei auf der Benutzung der Physik, den 3D-Animationen, sowie der Programmierung der Shader-Effekte

VERWENDETE BIBLIOTHEKEN

Grundsätzlich gilt, dass wenn möglich, vorhandene XNA-Funktionalitäten für das Projekt genutzt werden sollen.

Physik

Da die Simulation von physikalischen Objekten beim Spielprinzip eine große Rolle spielt, ist vorgesehen die Physik-Engine Jitter zu benutzen.
Jitter soll dabei sämtliche vom Spieler erzeugten Physik-Objekte (durch Trap-Functions bzw. einzelne Geschosse) als auch im Level bereits vorhandene Strukturen korrekt simulieren.

Ein Problempunkt ist dabei, dass Jitter über keine eigene Dokumentation verfügt, was unter Umständen zu Problemen führen kann. Aus diesem Grund muss im Falle von Problemen in Betracht gezogen werden, eine andere Physik-Engine zu verwenden (Havok, bzw. BEPU).

Sound

Zur besseren Simulation von Raumklängen soll die FMOD-Bibliothek eingesetzt werden, da 3D-Sound von XNA zwar unterstützt wird, aber einige Effekte (z.B Echo) nicht vorhanden sind.

Animationen

XNA erlaubt es, dass 3D-Modelle über Bones zu verfügen. Die eigentliche Boneanimation muss jedoch selbst implementiert werden.

Shader

Auch bei den Shadern gilt, dass die grundlegende Funktionalität in XNA zwar vorhanden ist, jedoch sehr eingeschränkt ist. Aus diesem Grund müssen sämtliche für das Projekt vorgesehene Shader-Effekte (Bloom, Normal-Mapping, etc.) von selbst implementiert werden.

VERWENDETE TOOLS UND DEREN INTEGRATION

Visual Studio

Visual Studio wird zur Programmierung des Projektes in C# eingesetzt.

Maya

Für die Modellierung sämtlicher Objekte, sowie die Animation des Spielers soll Maya benutzt werden. Ausgabeformat ist das FBX-Format

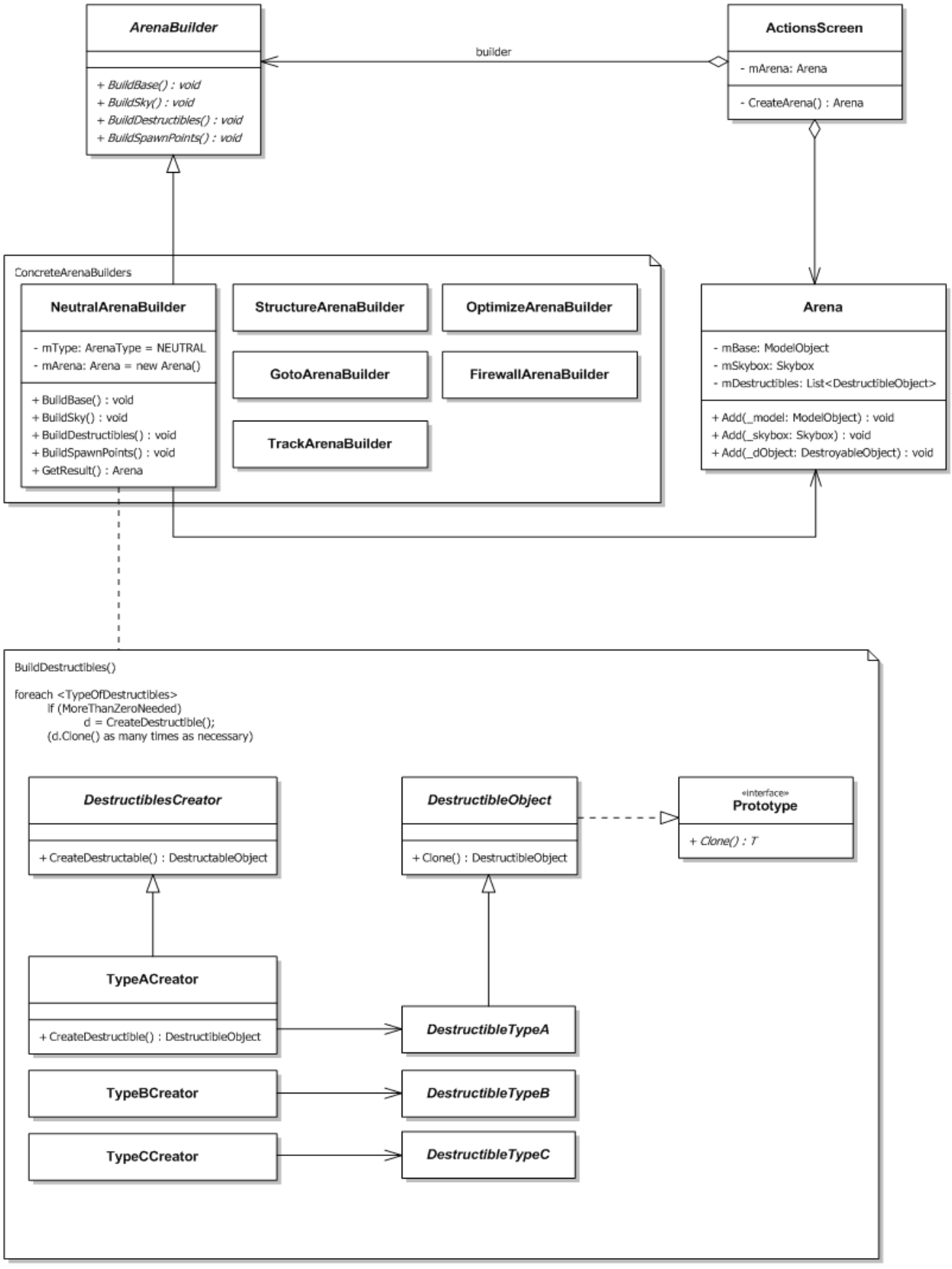
Bfxr

Bfxr soll für die Erstellung eigener Soundeffekte eingesetzt werden. Bfxr exportiert dabei ins WAVE-Format, welches auch von XNA unterstützt wird.

«interface»
ProgrammFactory
+ createShot() : shot
+ createTrap() : trap
+ createBoost() : boost
1
1
Arsenal
- curWeapon: WeaponType
- ammo: int[]
+ triggerCurShot() : void
+ triggerCurTrap() : void
+ triggerCurBoost() : void
+ changeWeaponTo(_t: WeaponType) : void
Player
Controls
+ ShootPressed : EventHandler
+ TrapPressed : EventHandler
+ BoostPressed : EventHandler
+ ChangeWPressed : EventH...
DeleteFactory
FirewallFactory
OptimizeFactory
TrackFactory
GotoFactory
+ createShot() : shot
+ createTrap() : trap
+ createBoost() : boost
Delete
DeleteShot
DeleteTrap
DeleteBoost
Firewall
FirewallShot
FirewallTrap
FirewallBoost
Optimize
OptimizeShot
OptimizeTrap
OptimizeBoost
Track
TrackShot
TrackTrap
TrackBoost
Goto
GotoShot
GotoTrap
GotoBoost
«interface»
Shot
«interface»
Trap
«interface»
Boost
PhysicalObject
PhysicsWorld

SOFTWAREARCHITEKTUR
Klassendiagramm des Waffen-Systems
Komponentenbeschreibung - GoF AbstractFactory

Der Player verfügt über eine Instanz der Klasse Arsenal, welche sein Waffensystem verwaltet. Sie enthält die verfügbare Munitionsmenge pro Programmtyp, sowie die aktuell ausgewählte Waffe. Sofern die Controls, welche den Input der Controller überwachen, das entsprechende Event feuern, kann die Waffe gewechselt oder in einem der drei Modi abgefeuert werden. Beim feuern wird ein Projektil von der entsprechenden Objekt Fabrik des Programms angefordert. Jede Ausprägung der Fabrik hat Zugriff auf einen eigenen Namespace mit Ausprägungen der jeweiligen Projektilkategorien. Innerhalb der Namespaces sind Informationen wie Modelle, Flugverhalten sowie Effekte verfügbar.
Jedem Projektil wird bei Erstellung ein PhysicalObject zugewiesen das der PhysicalWorld des Spiels hinzugefügt wird.


ArenaBuilder
+ BuildBase() : void
+ BuildSky() : void
+ BuildDestructibles() : void
+ BuildSpawnPoints() : void
builder
ActionsScreen
- mArena: Arena
- CreateArena() : Arena
ConcreteArenaBuilders
NeutralArenaBuilder
- mType: ArenaType = NEUTRAL
- mArena: Arena = new Arena()
+ BuildBase() : void
+ BuildSky() : void
+ BuildDestructibles() : void
+ BuildSpawnPoints() : void
+ GetResult() : Arena
StructureArenaBuilder
OptimizeArenaBuilder
GotoArenaBuilder
FirewallArenaBuilder
TrackArenaBuilder
Arena
- mBase: ModelObject
- mSkybox: Skybox
- mDestructibles: List<DestructibleObject>
+ Add(_model: ModelObject) : void
+ Add(_skybox: Skybox) : void
+ Add(_dObject: DestroyableObject) : void
BuildDestructibles()
foreach <TypeOfDestructibles>
If (MoreThanZeroNeeded)
d = CreateDestructible();
(d.Clone() as many times as necessary)
DestructiblesCreator
+ CreateDestructable() : DestructableObject
DestructibleObject
+ Clone() : DestructibleObject
«interface»
Prototype
+ Clone() : T
TypeACreator
+ CreateDestructible() : DestructibleObject
TypeBCreator
TypeCCreator
DestructibleTypeA
DestructibleTypeB
DestructibleTypeC

UseCase Diagramm der Ingame-Spieler-Aktionen

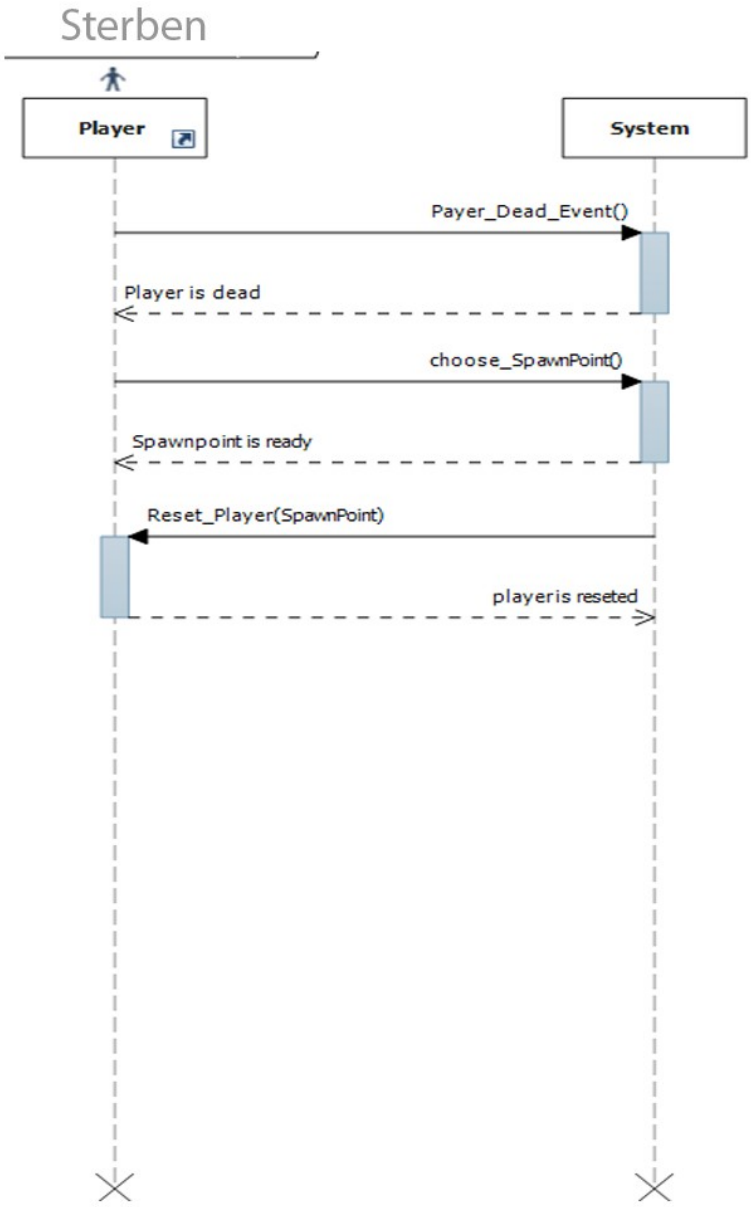
Klassendiagramm der Arenaerstellung

Komponentenbeschreibung - GoF Builder - GoF Factory - GoF Prototype

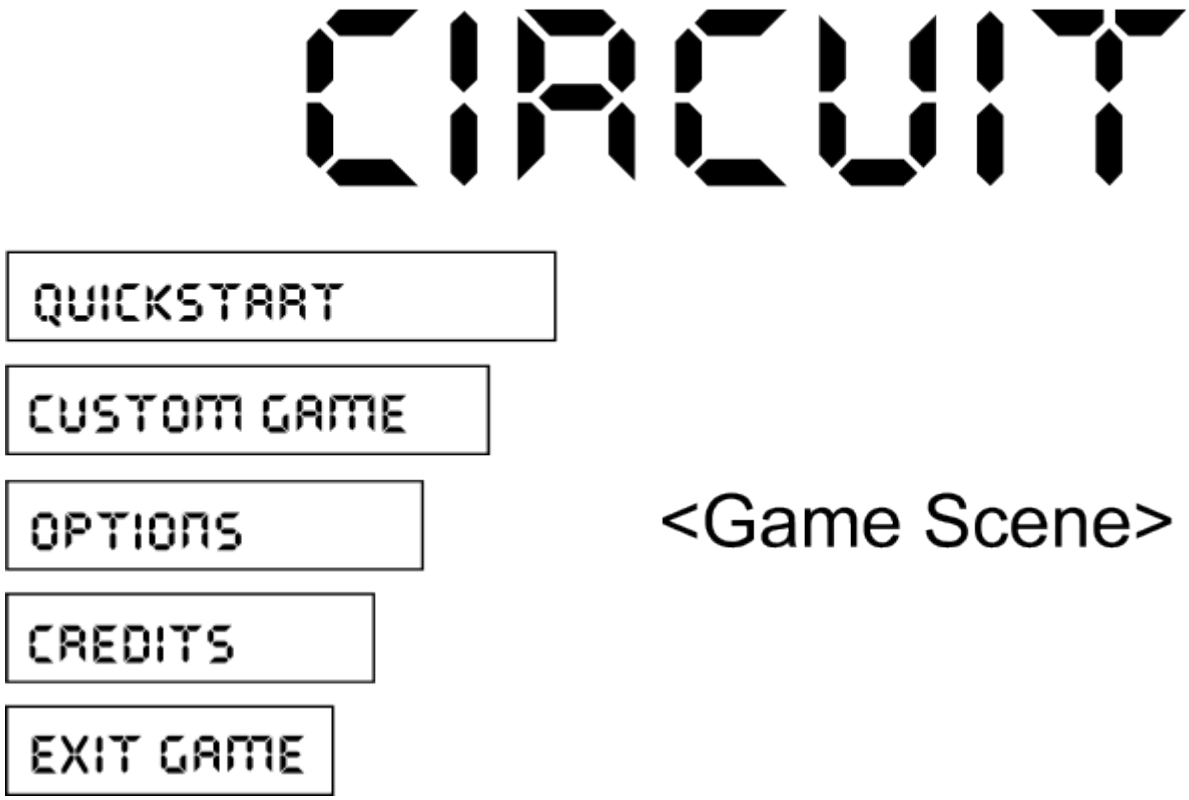
Der ActionScreen verfügt über eine Methode CreateArena() mittels der er zu Spielbeginn die Arena erstellt. Dies wird durch einen ArenaBuilder realisiert, der je nach angeforderten ArenaType eine dementsprechende Ausprägung hat. Der Builder erstellt zunächst eine leere Arena und fügt dieser ein Basis-Landschaftsobjekt, eine Skybox, DestructibleObjects etc. hinzu.

Innerhalb der BuildDestructibles() Methode des Builders wird die Arena mit zerstörbaren Objekten bestückt, welche je einen von drei Typen haben können. Hierzu wird zunächst eine Liste mit den vorgesehenen Positionen und Orientierungen der Objekte eingelesen. Nun wird für jeden Typ überprüft ob mindestens ein Objekt benötigt wird. Ist dies der Fall, erstellt ein entsprechender Creator eine Objektinstanz. Für jeden Listeneintrag wird zuletzt mittels der Clone() Methode des Prototype-Prinzips eine Kopie angefertigt und mit den gewünschten Parametern initialisiert.

Sequenz-Diagramm des Spielertodes



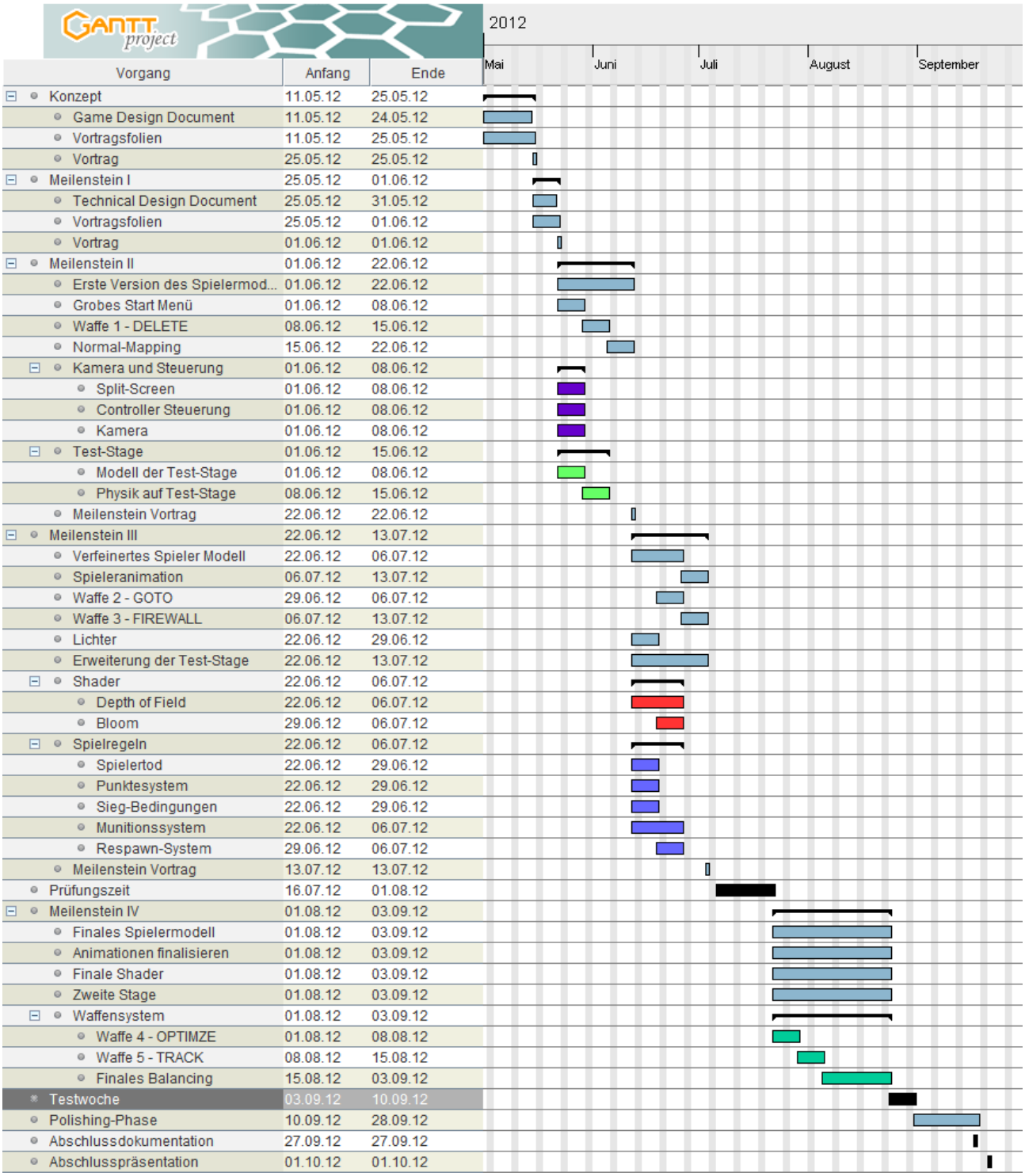
Mockup des Menu-Screens



TECHNISCHE MINDESTANFORDERUNGEN

- Das Spiel muss im Splitscreen mit 2 Spielern spielbar sein
- Es muss die gleichzeitige Nutzung von 2 Gamepads unterstützt werden
- Das Spiel muss mit mindestens 30 fps laufen

PROJEKTMANAGEMENT



Vorgang	Anfang	Ende
Konzept	11.05.12	25.05.12
Game Design Document	11.05.12	24.05.12
Vortragsfolien	11.05.12	25.05.12
Vortrag	25.05.12	25.05.12
Meilenstein I	25.05.12	01.06.12
Technical Design Document	25.05.12	31.05.12
Vortragsfolien	25.05.12	01.06.12
Vortrag	01.06.12	01.06.12
Meilenstein II	01.06.12	22.06.12
Erste Version des Spielermod...	01.06.12	22.06.12
Grobes Start Menü	01.06.12	08.06.12
Waffe 1 - DELETE	08.06.12	15.06.12
Normal-Mapping	15.06.12	22.06.12
Kamera und Steuerung	01.06.12	08.06.12
Split-Screen	01.06.12	08.06.12
Controller Steuerung	01.06.12	08.06.12
Kamera	01.06.12	08.06.12
Test-Stage	01.06.12	15.06.12
Modell der Test-Stage	01.06.12	08.06.12
Physik auf Test-Stage	08.06.12	15.06.12
Meilenstein Vortrag	22.06.12	22.06.12
Meilenstein III	22.06.12	13.07.12
Verfeinertes Spieler Modell	22.06.12	06.07.12
Spieleranimation	06.07.12	13.07.12
Waffe 2 - GOTO	29.06.12	06.07.12
Waffe 3 - FIREWALL	06.07.12	13.07.12
Lichter	22.06.12	29.06.12
Erweiterung der Test-Stage	22.06.12	13.07.12
Shader	22.06.12	06.07.12
Depth of Field	22.06.12	06.07.12
Bloom	29.06.12	06.07.12
Spielregeln	22.06.12	06.07.12
Spielertod	22.06.12	29.06.12
Punktesystem	22.06.12	29.06.12
Sieg-Bedingungen	22.06.12	29.06.12
Munitionssystem	22.06.12	06.07.12
Respawn-System	29.06.12	06.07.12
Meilenstein Vortrag	13.07.12	13.07.12
Prüfungszeit	16.07.12	01.08.12
Meilenstein IV	01.08.12	03.09.12
Finales Spielermodell	01.08.12	03.09.12
Animationen finalisieren	01.08.12	03.09.12
Finale Shader	01.08.12	03.09.12
Zweite Stage	01.08.12	03.09.12
Waffensystem	01.08.12	03.09.12
Waffe 4 - OPTIMZE	01.08.12	08.08.12
Waffe 5 - TRACK	08.08.12	15.08.12
Finales Balancing	15.08.12	03.09.12
Testwoche	03.09.12	10.09.12
Polishing-Phase	10.09.12	28.09.12
Abschlussdokumentation	27.09.12	27.09.12
Abschlusspräsentation	01.10.12	01.10.12