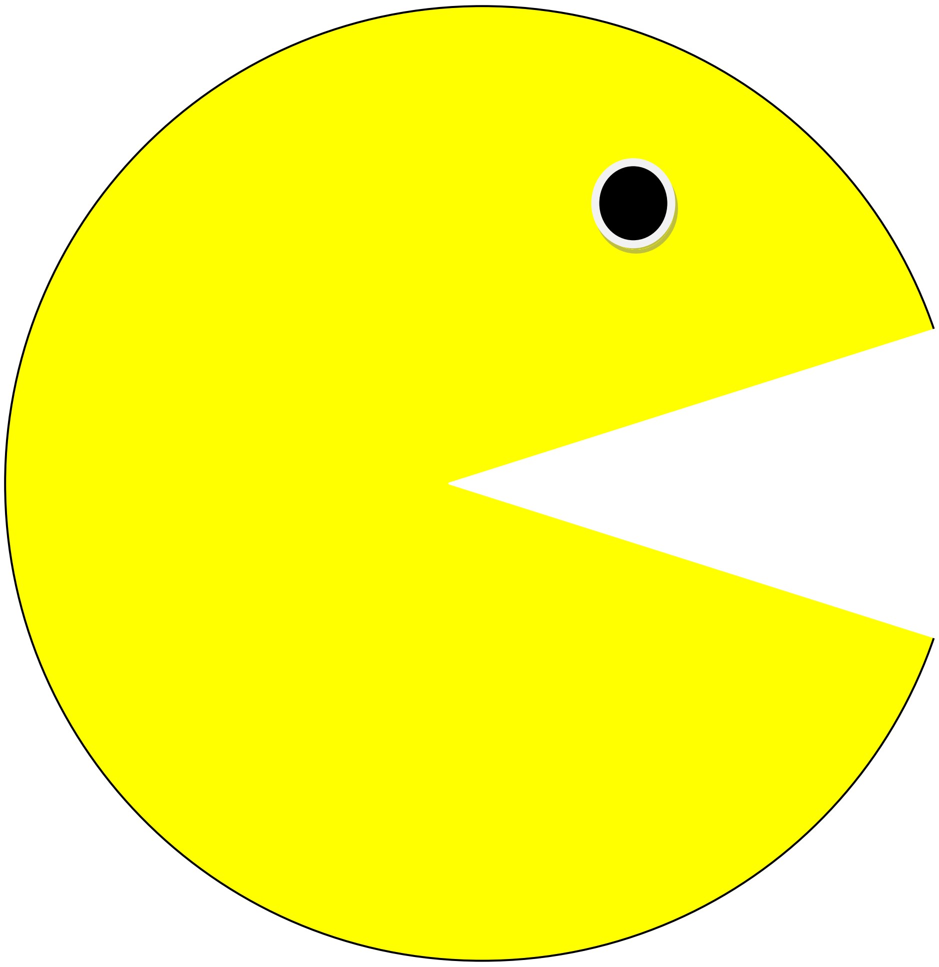
DV1435 Final Report

Thomas Sievert, Martin Säll, Lars Woxberg ,Kim Restad & Fredrik Johannesson



thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com

Contents

thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com,
kire10@student.bth.se, fredrik.johannesson@hotmail.com

Project description

Pacman::Reloaded is a 3D rendition of the timeless classic Pacman.

The game starts with a title screen, where there are four options: Play the game, view the highscore, view the credits, and quit the game. The game has a set number of stages for Pacman to go through. When he has finished them all, he is sent back to the first stage, on a slightly harder difficulty. Thus, the game goes on indefinitely, or until Pacman dies. When the game is over, the player might be registered to the highscore list. Pacman::Reloaded is developed on Windows for Windows PCs with Direct3D10 compatibility.

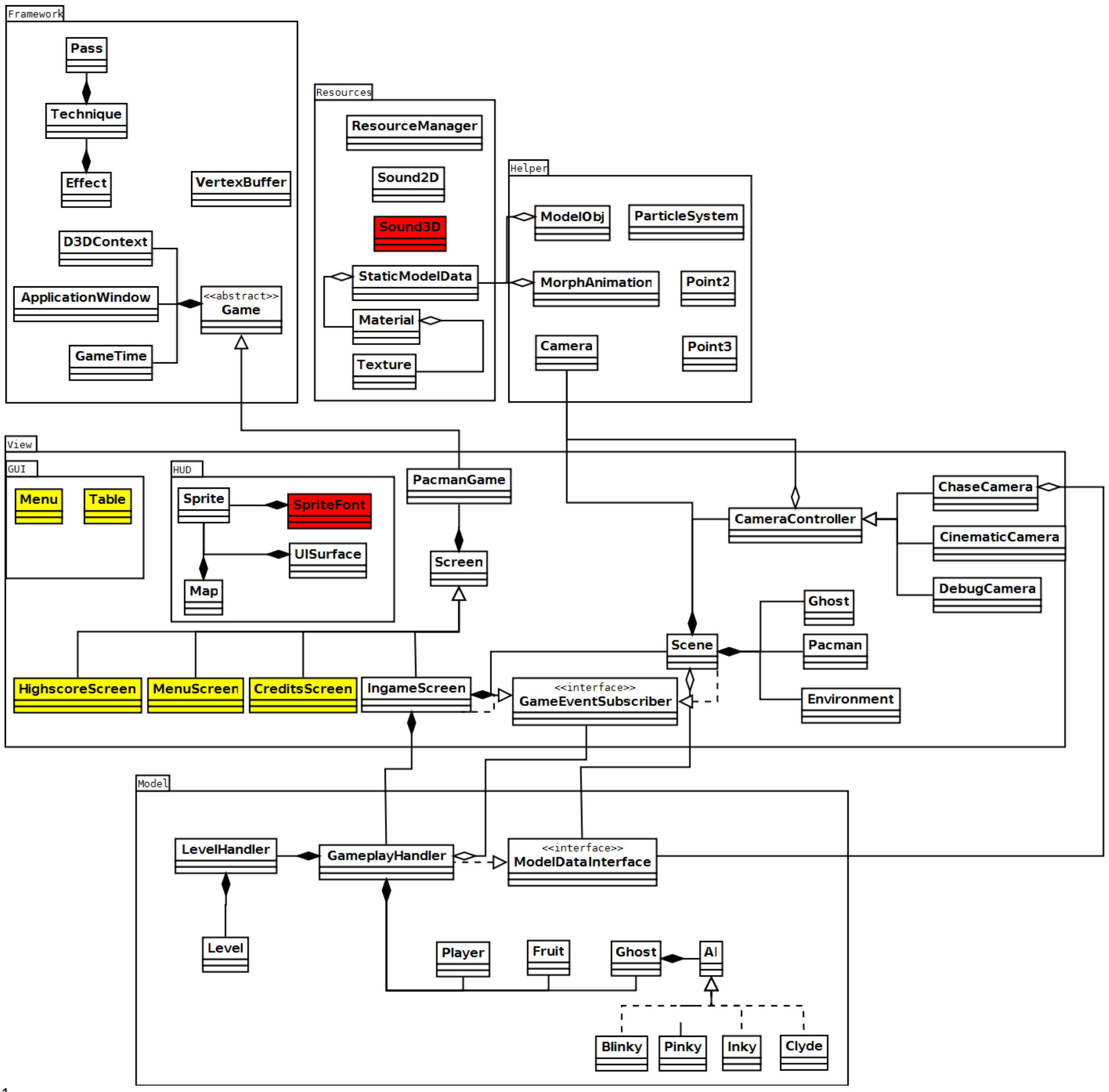
Each level has a certain amount of pellets spread out, and the goal is for Pacman to eat all of them while at the same time avoiding the dreaded ghosts, chasing him around. The layout of the level is tile-based and maze-like, and the density of pellets is high. Wherever there isn't a wall tile, there will initially reside a pellet. The only exception to this is the ghosts' home.

Pacman moves around in a simple manner: in one of four directions. He can turn corners, or turn around on the spot, but he can never stand still. The same rules apply to the ghosts. Whenever Pacman finds himself on a tile where there is a pellet, he eats it, and gains some points. If, however, he finds himself on the same tile as a ghost, he dies.

There are a few special pellets, called "power pellets" that make Pacman invincible and able to eat his enemies for a limited period of time. There may also appear a fruit occasionally, which gives Pacman a large amount of points. If he has not eaten all the pellets within a certain period of time, Pacman dies. The game goes on until he has died three times. If the player has managed to collect a high enough score, he can enter his name on the highscore board.

Architecture overview

The following diagram describes the system's final design. The red components are not yet implemented, and the yellow components are unfinished.



[1]

[1] ClassOverviewDiagram.png

thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com

Description of components

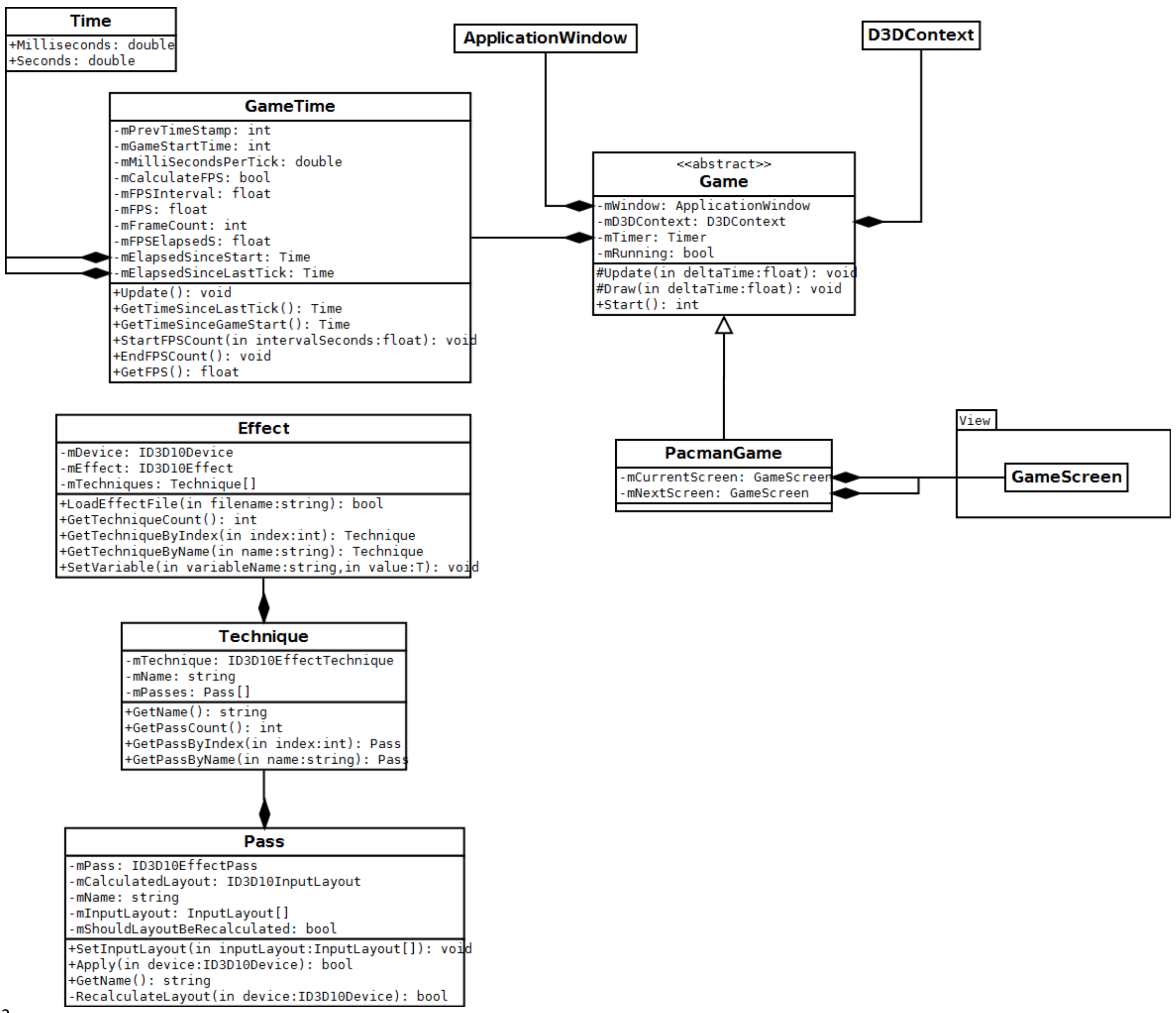
The game is designed very much with a Model/View setup in mind. As can be seen in the diagram above (*ClassOverviewDiagram.png*), the Model package is completely independent, only exposing an interface for the data and the events relevant to the view. The View package also functions as a controller, updating the Model from the Ingame screen component.

The Helper, Framework and Resource packages contain the "basics" and are as such accessed by both the View and the Model, when need arises. We also regard them as less relevant to explain in full detail, since they don't describe the game mechanics as much as they are a foundation.

Framework

Handles the basic setup of the game, such as creating a window, initializing Direct3D, and provides an abstraction to their services.

- Game
  Wraps around the game loop. Is meant to be extended by the main game class.
- D3DContext
  Abstracts Direct3D functionality.
- ApplicationWindow
  Abstracts Win32 functionality for handling windows.
- Timer
  A class for measuring time with high precision.
- VertexBuffer
  An abstraction for the data structure used to send information to the GPU.
- Effect
  An abstraction to handle effect (.fx) files, i.e. shaders.
  - Pass
    A subcomponent to the Technique class. One Technique may consist of many passes.
  - Technique
    A subcomponent to the Effect class. One Effect may consist of many techniques.


Time
+Milliseconds: double
+Seconds: double
ApplicationWindow
D3DContext
GameTime
-mPrevTimeStamp: int
-mGameStartTime: int
-mMilliSecondsPerTick: double
-mCalculateFPS: bool
-mFPSInterval: float
-mFPS: float
-mFrameCount: int
-mFPSElapsedS: float
-mElapsedSinceStart: Time
-mElapsedSinceLastTick: Time
+Update(): void
+GetTimeSinceLastTick(): Time
+GetTimeSinceGameStart(): Time
+StartFPSCount(in intervalSeconds:float): void
+EndFPSCount(): void
+GetFPS(): float
<<abstract>>
Game
-mWindow: ApplicationWindow
-mD3DContext: D3DContext
-mTimer: Timer
-mRunning: bool
#Update(in deltaTime:float): void
#Draw(in deltaTime:float): void
+Start(): int
Effect
-mDevice: ID3D10Device
-mEffect: ID3D10Effect
-mTechniques: Technique[]
+LoadEffectFile(in filename:string): bool
+GetTechniqueCount(): int
+GetTechniqueByIndex(in index:int): Technique
+GetTechniqueByName(in name:string): Technique
+SetVariable(in variableName:string,in value:T): void
PacmanGame
-mCurrentScreen: GameScreen
-mNextScreen: GameScreen
View
GameScreen
Technique
-mTechnique: ID3D10EffectTechnique
-mName: string
-mPasses: Pass[]
+GetName(): string
+GetPassCount(): int
+GetPassByIndex(in index:int): Pass
+GetPassByName(in name:string): Pass
Pass
-mPass: ID3D10EffectPass
-mCalculatedLayout: ID3D10InputLayout
-mName: string
-mInputLayout: InputLayout[]
-mShouldLayoutBeRecalculated: bool
+SetInputLayout(in inputLayout:InputLayout[]): void
+Apply(in device:ID3D10Device): bool
+GetName(): string
-RecalculateLayout(in device:ID3D10Device): bool
[2]

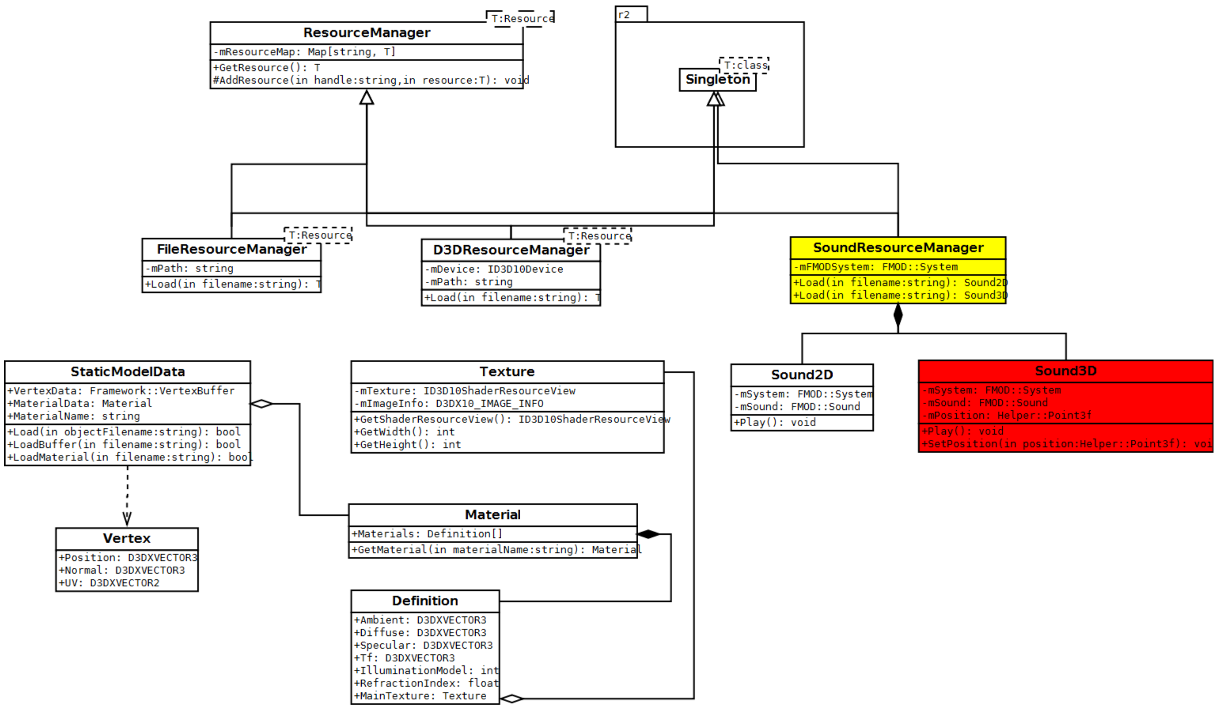
The Framework package consists of components abstracting Win32 and Direct3D functionality. These make up the very foundation upon which the game is run. Much of it is boilerplate work that has very little impact on the actual design.

[2] FrameworkClassDiagram.png

Resources

Manages all resources we do not wish to load twice.

- ResourceManager
  Holds references to resources of a specific type. This is a templated singleton class. Thanks to this, resources are easily available in any part of the project, should they be needed.
- Sound2D
  Plays a sound.
- Sound3D
  Plays a sound from a certain position in a 3D scene.
- StaticModelData
  A struct-like class that contains the mesh data for a 3D object.
- Material
  Holds the material information for a 3D object.
- Texture
  Holds a texture for a 3D object.



[3]

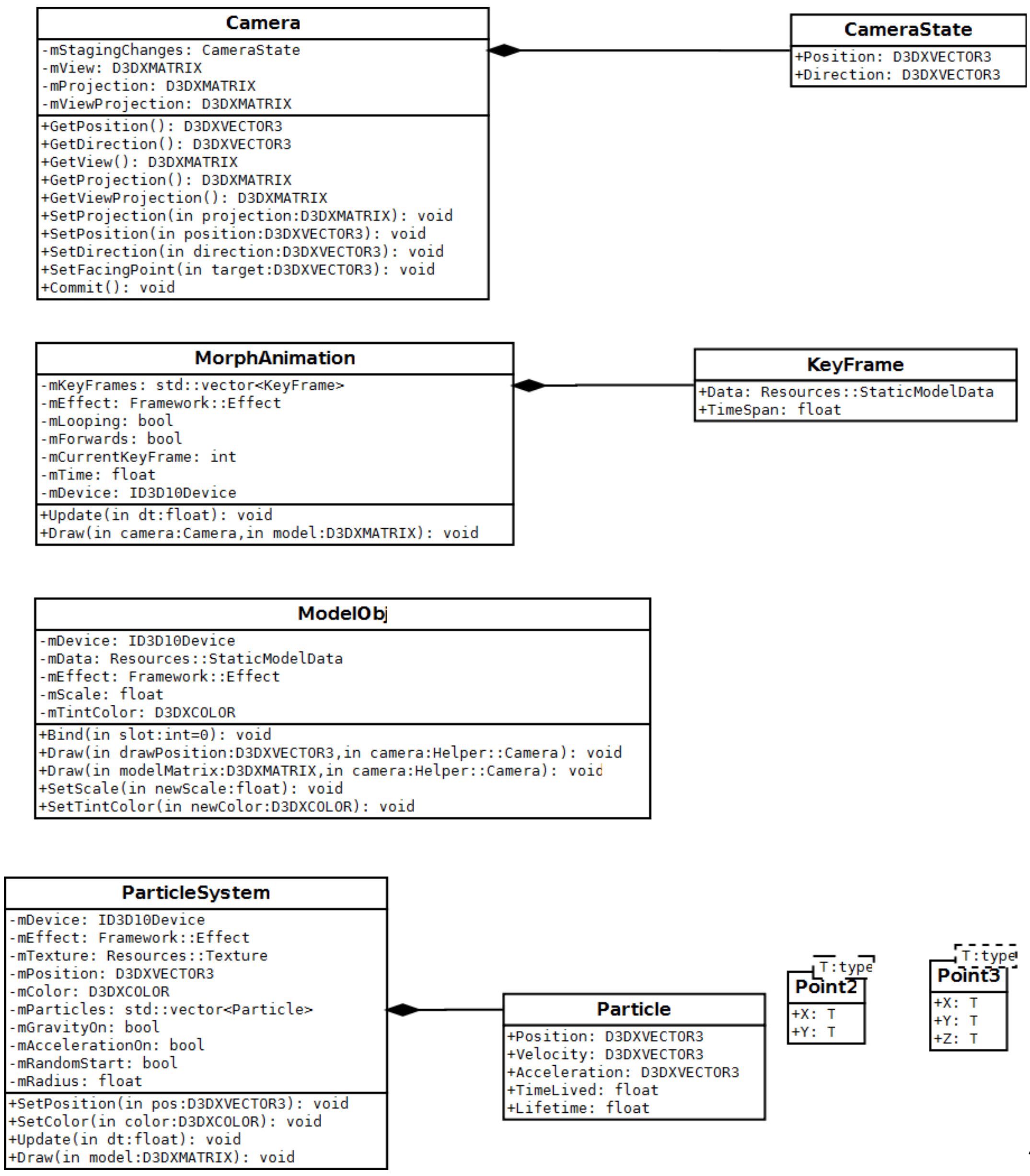
The resources package contains all components concerning the game's various resources. A Singleton pattern is used for the various Resource Managers to make the resources easily accessible from all parts of the system. The Singleton base class from which the managers are derived comes from an open source third party library called r2tk.

[3] ResourceClassDiagram.png

Helper

Classes with generic use, needed throughout the system.

- MorphAnimation
  A collection of 3D models, using morph animation to interpolate between key frames.
- ModelObj
  A representation of a 3D model from an .obj file.
- ParticleSystem
  A system of particles used to create various effects.
- Camera
  A camera used to look at a 3D scene.



Camera
-mStagingChanges: CameraState
-mView: D3DXMATRIX
-mProjection: D3DXMATRIX
-mViewProjection: D3DXMATRIX
+GetPosition(): D3DXVECTOR3
+GetDirection(): D3DXVECTOR3
+GetView(): D3DXMATRIX
+GetProjection(): D3DXMATRIX
+GetViewProjection(): D3DXMATRIX
+SetProjection(in projection:D3DXMATRIX): void
+SetPosition(in position:D3DXVECTOR3): void
+SetDirection(in direction:D3DXVECTOR3): void
+SetFacingPoint(in target:D3DXVECTOR3): void
+Commit(): void
CameraState
+Position: D3DXVECTOR3
+Direction: D3DXVECTOR3
MorphAnimation
-mKeyFrames: std::vector<KeyFrame>
-mEffect: Framework::Effect
-mLooping: bool
-mForwards: bool
-mCurrentKeyFrame: int
-mTime: float
-mDevice: ID3D10Device
+Update(in dt:float): void
+Draw(in camera:Camera,in model:D3DXMATRIX): void
KeyFrame
+Data: Resources::StaticModelData
+TimeSpan: float
ModelObj
-mDevice: ID3D10Device
-mData: Resources::StaticModelData
-mEffect: Framework::Effect
-mScale: float
-mTintColor: D3DXCOLOR
+Bind(in slot:int=0): void
+Draw(in drawPosition:D3DXVECTOR3,in camera:Helper::Camera): void
+Draw(in modelMatrix:D3DXMATRIX,in camera:Helper::Camera): void
+SetScale(in newScale:float): void
+SetTintColor(in newColor:D3DXCOLOR): void
ParticleSystem
-mDevice: ID3D10Device
-mEffect: Framework::Effect
-mTexture: Resources::Texture
-mPosition: D3DXVECTOR3
-mColor: D3DXCOLOR
-mParticles: std::vector<Particle>
-mGravityOn: bool
-mAccelerationOn: bool
-mRandomStart: bool
-mRadius: float
+SetPosition(in pos:D3DXVECTOR3): void
+SetColor(in color:D3DXCOLOR): void
+Update(in dt:float): void
+Draw(in model:D3DXMATRIX): void
Particle
+Position: D3DXVECTOR3
+Velocity: D3DXVECTOR3
+Acceleration: D3DXVECTOR3
+TimeLived: float
+Lifetime: float
T:type
Point2
+X: T
+Y: T
T:type
Point3
+X: T
+Y: T
+Z: T
[4]

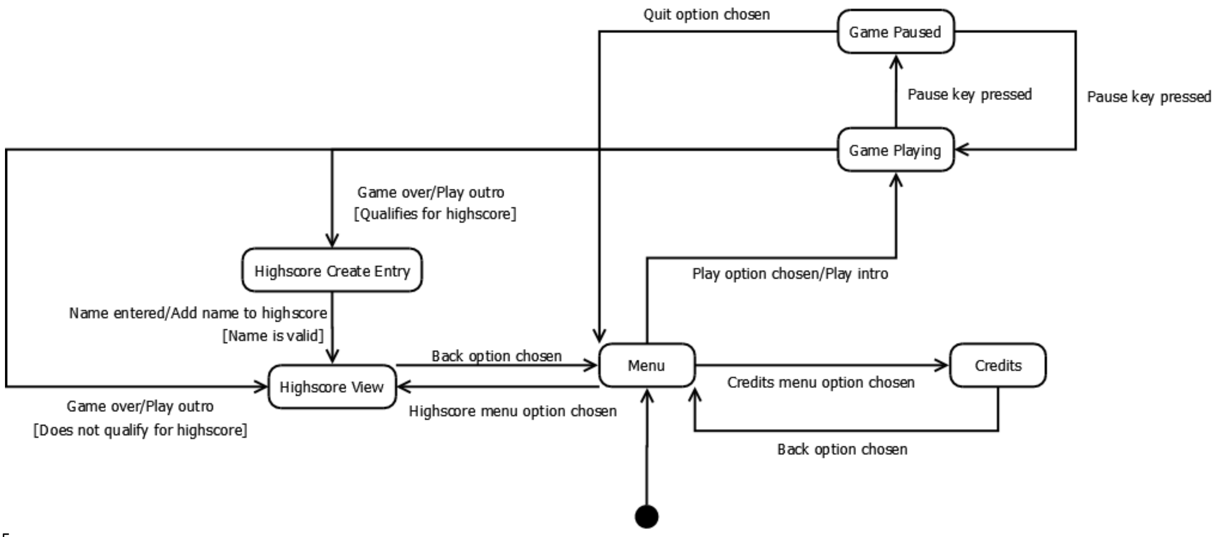
The Helper package is a small package providing various primitives and abstractions that make things a lot easier in other packages.

[4] HelperClassDiagram.png

View

Classes that are concerned with outputting data to the user.

- PacmanGame
  Main game class, runs the actual game loop.
- GameEventSubscriber
  An interface used in the model package to notify the view of certain game-specific events.
- Screen
  The base class for the different game screens.
  - MenuScreen
    The main menu, where the game starts.
  - HighscoreScreen
    The highscore screen, meant to present the highest scores of past players. Can also insert a new entry into the list of scores.
  - CreditsScreen
    A screen mentioning the development team.
  - IngameScreen
    The screen where most of the game happens. Draws the scene and updates the model.



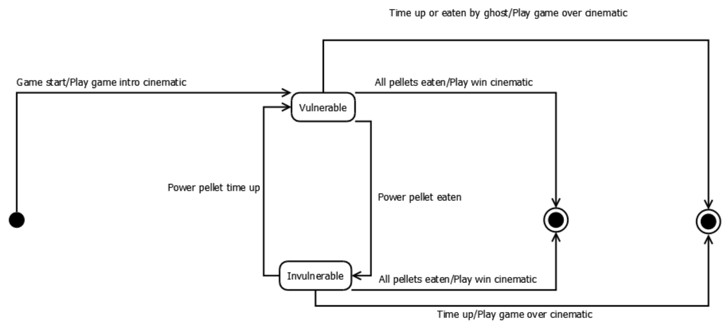
[5]

- GUI
  Components that make up the graphical user interface in the screens.
  - Menu
    A list of selectable options.
  - Table
    A table of data.
- HUD
  Contains classes for rendering 2D graphics and different 2D game elements.

[5] UIStateMachine.png

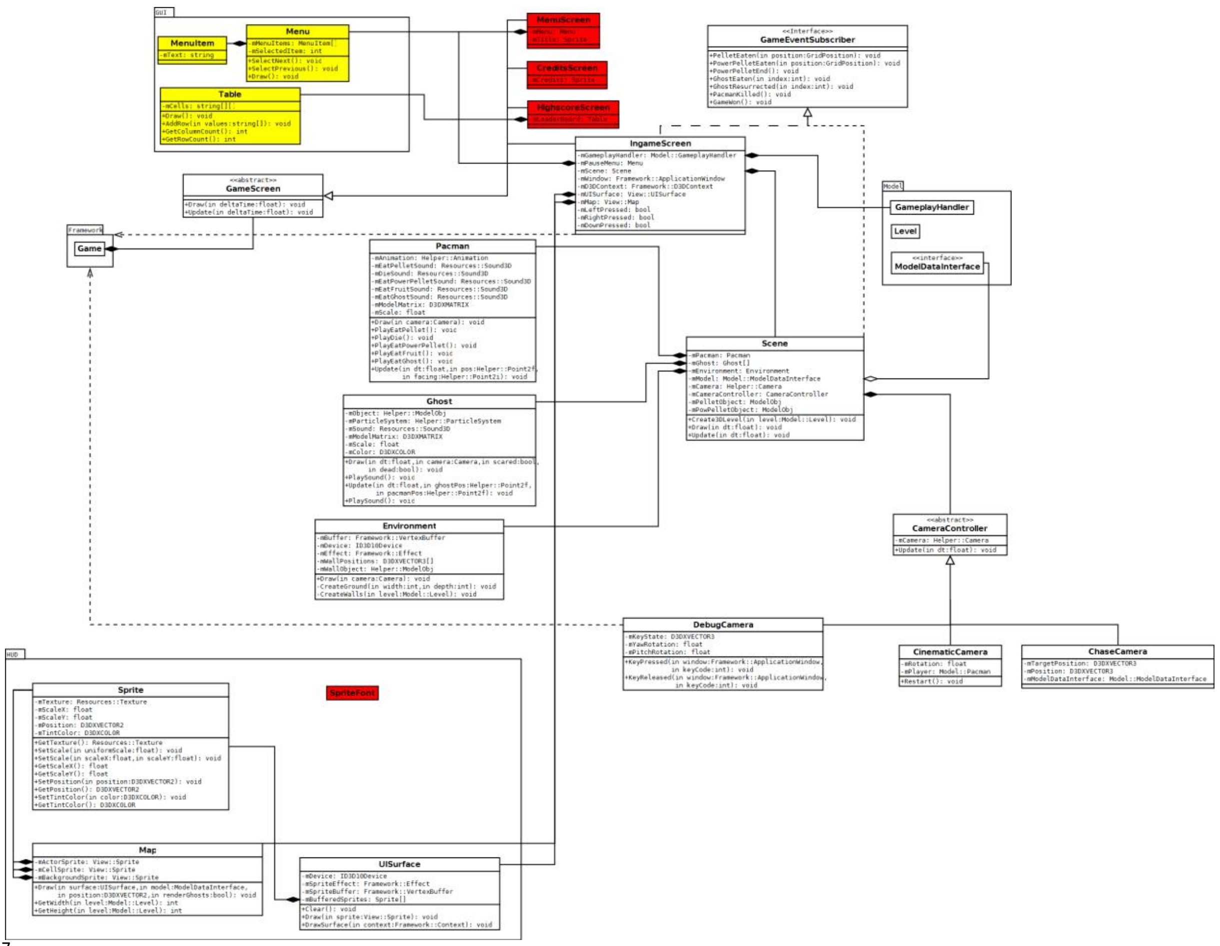
- Sprite
  A sprite is a wrapper around a texture, for rendering 2D images on screen.
- SpriteFont
  A collection of sprites of characters in a font, for rendering text.
- UISurface
  A UISurface transforms sprites to the screen and allows rendering sprites within a defined area.
- Map
  A map of the game, showing the level layout, where you are and optionally where the ghosts are.

- Scene
  The component that draws the actual game. Collects data from the model and presents it graphically.
- Environment
  Draws the tiles of the level, walls and floors.
- Ghost
  Draws a ghost, with an model and a particle system.
- Pacman
  Draws Pacman, with an animation.


Time up or eaten by ghost/Play game over cinematic
Game start/Play game intro cinematic
All pellets eaten/Play win cinematic
Vulnerable
Power pellet time up
Power pellet eaten
Invulnerable
All pellets eaten/Play win cinematic
Time up/Play game over cinematic
[6]

- CameraController
  A strategy pattern for different types of camera movement.
  - ChaseCamera
    A camera controller, chases Pacman's position, and can be turned to look either forwards or backwards.
  - CinematicCamera
    A camera controller designed to circle around Pacman.
  - DebugCamera
    A camera controller designed to move the camera around freely.

[6] PacmanStateMachine.png



7

The View package is the largest package in the system, responsible for all output to the user. This includes 2D rendering such as GUI and HUD elements, 3D rendering such as the actual Pacman game, and sound output, both in two and three dimensions.

The View package communicates with the Model package using the Observer pattern through the GameEventSubscriber interface, and also collects the remaining data necessary from an interface in Model called ModelDataInterface.

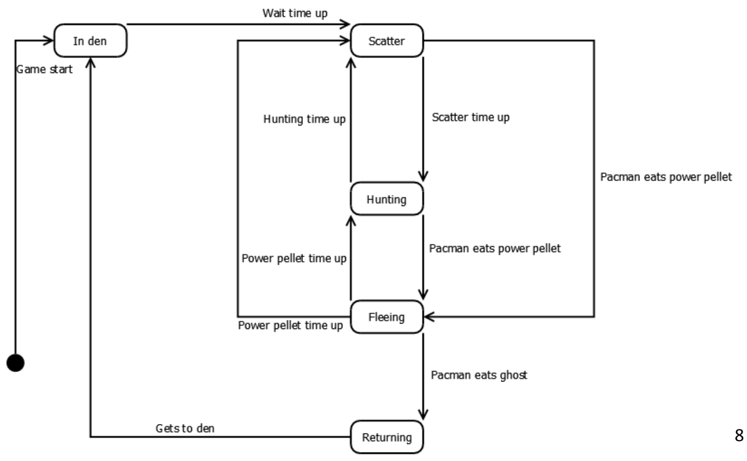
To control the camera the Strategy pattern is used, since there are three very specific behaviors a camera can have.

7 ViewClassDiagram.png

Model

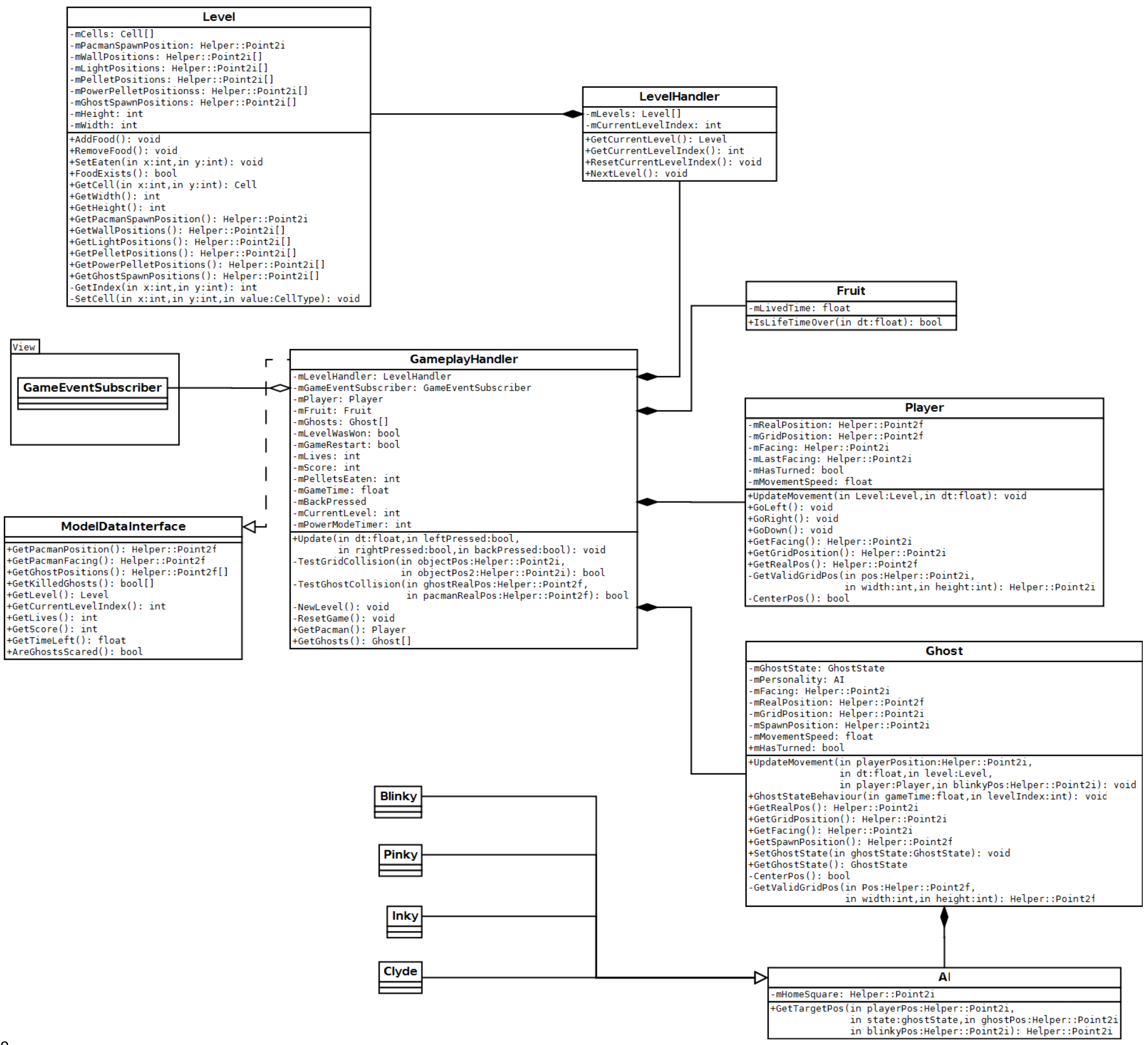
Contains the classes concerning the logic of a Pacman game session.

- GameplayHandler
  Handles one session, moving through all levels until game over.
- ModelDataInterface
  Exposes data to the view, necessary to render the game.
- LevelHandler
  Handles the progression through the levels.
- Level
  Contains the data of one level. Is able to load itself from a .png file.
- Player
  Handles the movement and logic of Pacman.
- Fruit
  Handles the fruit bonuses that sometimes appear in the game, with a timer.
- Ghost
  Handles the logic of the enemies in the game. Contains an AI object, which can be used to change the behavior with the strategy pattern.
- AI
  Determines the behavior of a ghost. The following AIs will be implemented, one for each ghost:
  - Blinky
    Moves towards Pacman's actual position.
  - Pinky
    Attempts to move in front of Pacman.
  - Inky
    Tries to move to its mirror position in regards to Pacman's path.
  - Clyde
    Moves towards Pacman, but flees when he gets too close.



[8]

[8] EnemyStateMachine.png


Level
-mCells: Cell[]
-mPacmanSpawnPosition: Helper::Point2i
-mWallPositions: Helper::Point2i[]
-mLightPositions: Helper::Point2i[]
-mPelletPositions: Helper::Point2i[]
-mPowerPelletPositionss: Helper::Point2i[]
-mGhostSpawnPositions: Helper::Point2i[]
-mHeight: int
-mWidth: int
+AddFood(): void
+RemoveFood(): void
+SetEaten(in x:int,in y:int): void
+FoodExists(): bool
+GetCell(in x:int,in y:int): Cell
+GetWidth(): int
+GetHeight(): int
+GetPacmanSpawnPosition(): Helper::Point2i
+GetWallPositions(): Helper::Point2i[]
+GetLightPositions(): Helper::Point2i[]
+GetPelletPositions(): Helper::Point2i[]
+GetPowerPelletPositions(): Helper::Point2i[]
+GetGhostSpawnPositions(): Helper::Point2i[]
-GetIndex(in x:int,in y:int): int
-SetCell(in x:int,in y:int,in value:CellType): void
LevelHandler
-mLevels: Level[]
-mCurrentLevelIndex: int
+GetCurrentLevel(): Level
+GetCurrentLevelIndex(): int
+ResetCurrentLevelIndex(): void
+NextLevel(): void
Fruit
-mLivedTime: float
+IsLifeTimeOver(in dt:float): bool
View
GameEventSubscriber
GameplayHandler
-mLevelHandler: LevelHandler
-mGameEventSubscriber: GameEventSubscriber
-mPlayer: Player
-mFruit: Fruit
-mGhosts: Ghost[]
-mLevelWasWon: bool
-mGameRestart: bool
-mLives: int
-mScore: int
-mPelletsEaten: int
-mGameTime: float
-mBackPressed
-mCurrentLevel: int
-mPowerModeTimer: int
+Update(in dt:float,in leftPressed:bool, in rightPressed:bool,in backPressed:bool): void
-TestGridCollision(in objectPos:Helper::Point2i, in objectPos2:Helper::Point2i): bool
-TestGhostCollision(in ghostRealPos:Helper::Point2f, in pacmanRealPos:Helper::Point2f): bool
-NewLevel(): void
-ResetGame(): void
+GetPacman(): Player
+GetGhosts(): Ghost[]
ModelDataInterface
+GetPacmanPosition(): Helper::Point2f
+GetPacmanFacing(): Helper::Point2f
+GetGhostPositions(): Helper::Point2f[]
+GetKilledGhosts(): bool[]
+GetLevel(): Level
+GetCurrentLevelIndex(): int
+GetLives(): int
+GetScore(): int
+GetTimeLeft(): float
+AreGhostsScared(): bool
Player
-mRealPosition: Helper::Point2f
-mGridPosition: Helper::Point2f
-mFacing: Helper::Point2i
-mLastFacing: Helper::Point2i
-mHasTurned: bool
-mMovementSpeed: float
+UpdateMovement(in Level:Level,in dt:float): void
+GoLeft(): void
+GoRight(): void
+GoDown(): void
+GetFacing(): Helper::Point2i
+GetGridPosition(): Helper::Point2i
+GetRealPos(): Helper::Point2f
-GetValidGridPos(in pos:Helper::Point2i, in width:int,in height:int): Helper::Point2i
-CenterPos(): bool
Ghost
-mGhostState: GhostState
-mPersonality: AI
-mFacing: Helper::Point2i
-mRealPosition: Helper::Point2f
-mGridPosition: Helper::Point2i
-mSpawnPosition: Helper::Point2i
-mMovementSpeed: float
+mHasTurned: bool
+UpdateMovement(in playerPosition:Helper::Point2i, in dt:float,in level:Level, in player:Player,in blinkyPos:Helper::Point2i): void
+GhostStateBehaviour(in gameTime:float,in levelIndex:int): void
+GetRealPos(): Helper::Point2i
+GetGridPosition(): Helper::Point2i
+GetFacing(): Helper::Point2i
+GetSpawnPosition(): Helper::Point2f
+SetGhostState(in ghostState:GhostState): void
+GetGhostState(): GhostState
-CenterPos(): bool
-GetValidGridPos(in Pos:Helper::Point2f, in width:int,in height:int): Helper::Point2f
Blinky
Pinky
Inky
Clyde
AI
-mHomeSquare: Helper::Point2i
+GetTargetPos(in playerPos:Helper::Point2i, in state:ghostState,in ghostPos:Helper::Point2i in blinkyPos:Helper::Point2i): Helper::Point2i

[9]

The Model package is where the game is run. Due to the crisp distinction between the View and Model packages, this package could essentially be reused with a completely different view without any modifications. Naturally, the reverse should also hold true. The Gameplay Handler component calls certain methods in its GameEventSubscribers when certain events occur that require the View to display particular effects, otherwise it supplies the View related information by implementing the Model Data Interface.

For the ghost AIs, Strategy pattern has been chosen. The difference between the ghosts is how they decide their target position. That one method is what separates them, so a Strategy pattern is tailor made for this.

[9] ModelClassDiagram.png

thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com

Features not implemented

The features listed in the project description that we have not implemented fully are the following (along with suggestions on how to implement them according to our design):

- Audio
  Audio components are already incorporated in the design.
- Levels with several floors
  No changes in the design would be necessary, since the format for a level has room to accommodate it. Another tile would have to be added, and so an extra method in the ModelDataInterface to reach it would follow.
- Several light sources
  The design supports "a couple" of light sources, but more than that would require us to change the rendering technique to deferred rendering to achieve anything close to playable performance.
- Pacman being packing
  We would have added a weapon base class, and a strategy pattern to enable different kinds of weapons. Just as with the extra tile in multi-floor levels extra methods would need to be added to communicate the bullets' positions to the view.

Evaluation

While the end product is not entirely complete, we are very content with the design. The Model/View/Controller design turned out well. The game's logic is completely separated from the graphics, which led to a degree of modularity in the project. Connecting the view and model packages did not prove to be much of a problem, despite their nigh-complete lack of coupling.

It would have been desirable to optimize the game a bit more - on slower systems it does not run at full speed - however, such advanced algorithms would have required a lot of time both to learn and implement.

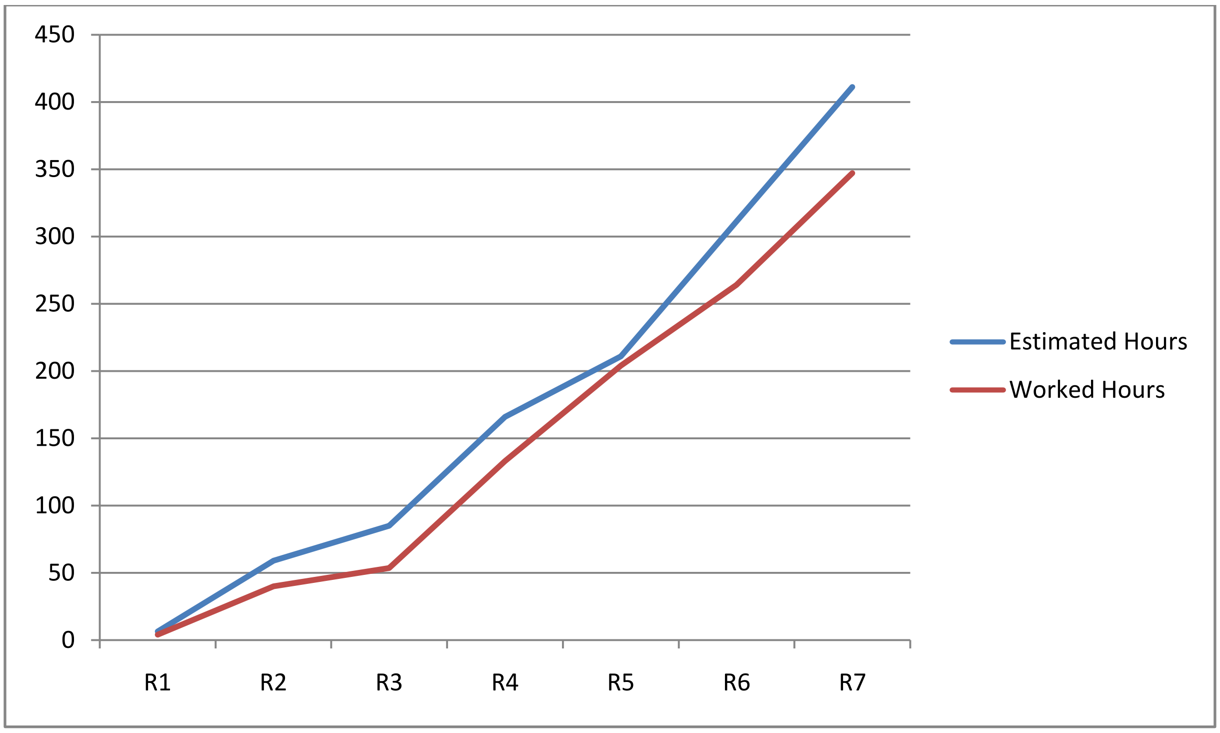
During the course of the project we encountered the following main problems (along with what we did to try and solve them):

- Time
  Time remained the biggest problem throughout the entire project. We had difficulties in estimating the time a task would take, which led to problems regarding the WBS as tasks got finished too fast or did not get finished on time. On top of that we rarely spent the supposed amount of time every week, partly due to the WBS confusion. There was not much to do about this, than to expect it, and plan ahead. In our plan we ended up with a "buffer week", of which we had great use as time started slipping away. We also had to remove some of our fancier features (such as deferred rendering, render batch, scene graph, etc) because they would have taken too much time.
- Project management
  We spent the first half of the project without a project management software, which made making reports take a lot more time than necessary. Installing and setting up Redmine may have taken a lot of time, but it made our lives so much easier once we had it running.
- Metrics
  It took us quite some time to figure out which product metrics to present, and even now we are not sure about them. Eventually we wrote a small Python script that can count lines, source lines and commented lines of code for us, and we decided to present the amount of finished tasks versus the total amount of tasks. While not very sharp measurements, they still give somewhat of an overview of our progress.

All these problems (along with ones too small to mention) got resolved in one way or another. What we have, in the end, is a product near completion, which we value a lot higher than a more unfinished, pre-optimized product.

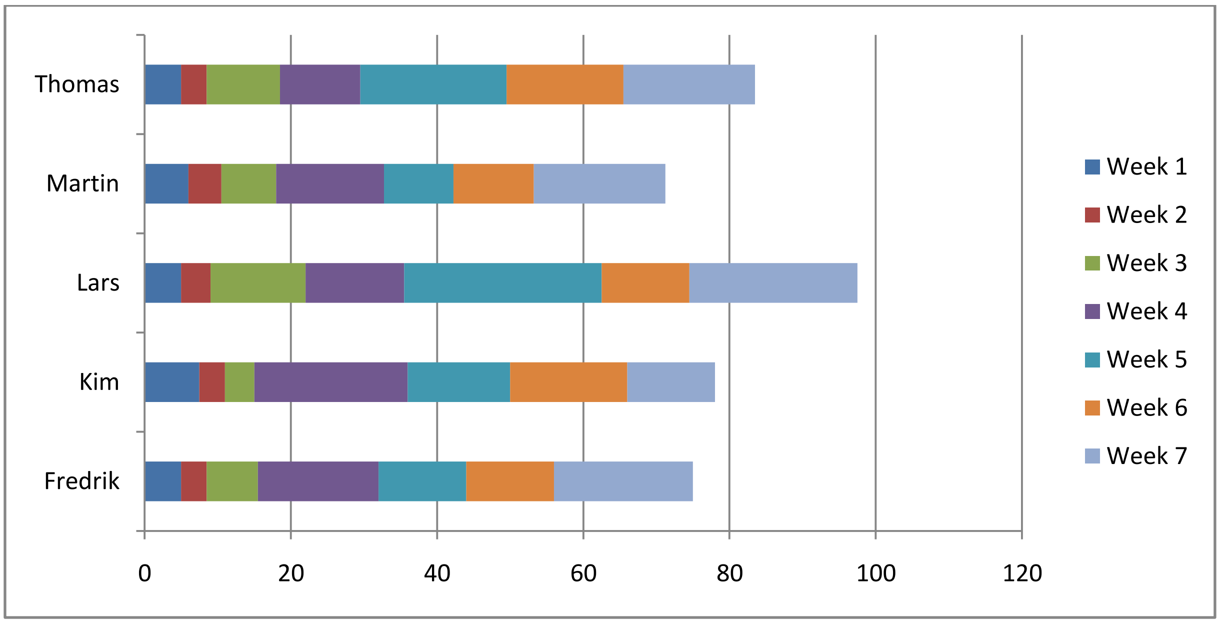
Progress

Below is a diagram displaying the project's schedule compliance. The horizontal axis represents each week. The vertical axis represents time. The lines represent the cumulative estimated and worked hours, respectively.



Resources spent

Below is a diagram displaying the resources (i.e. hours) spent, per person, per week.

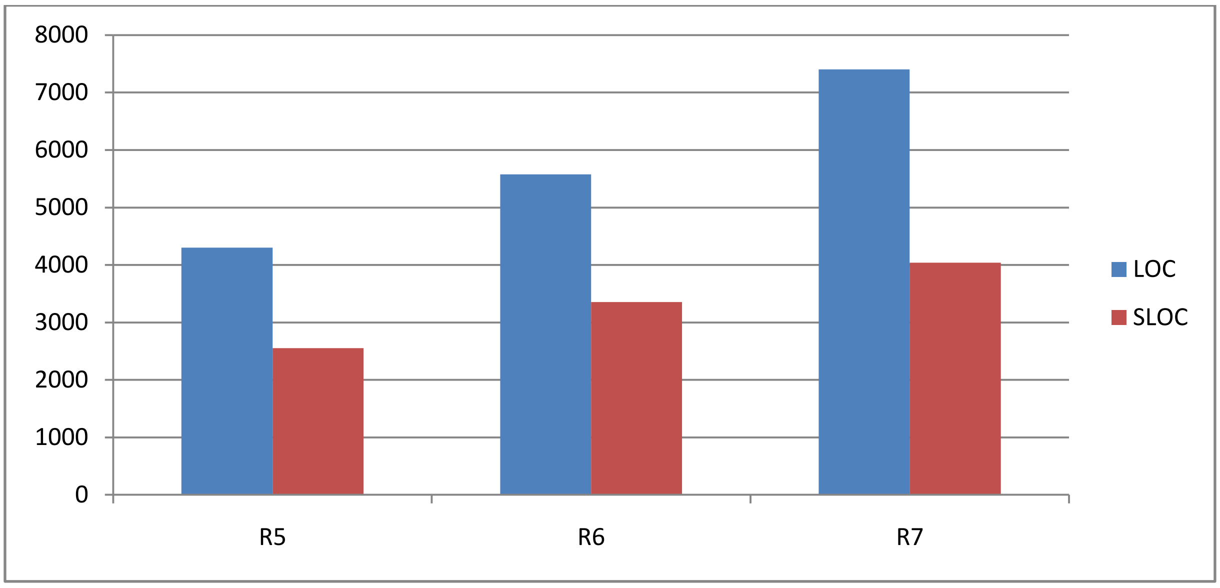


thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com

Product metrics

Code metrics

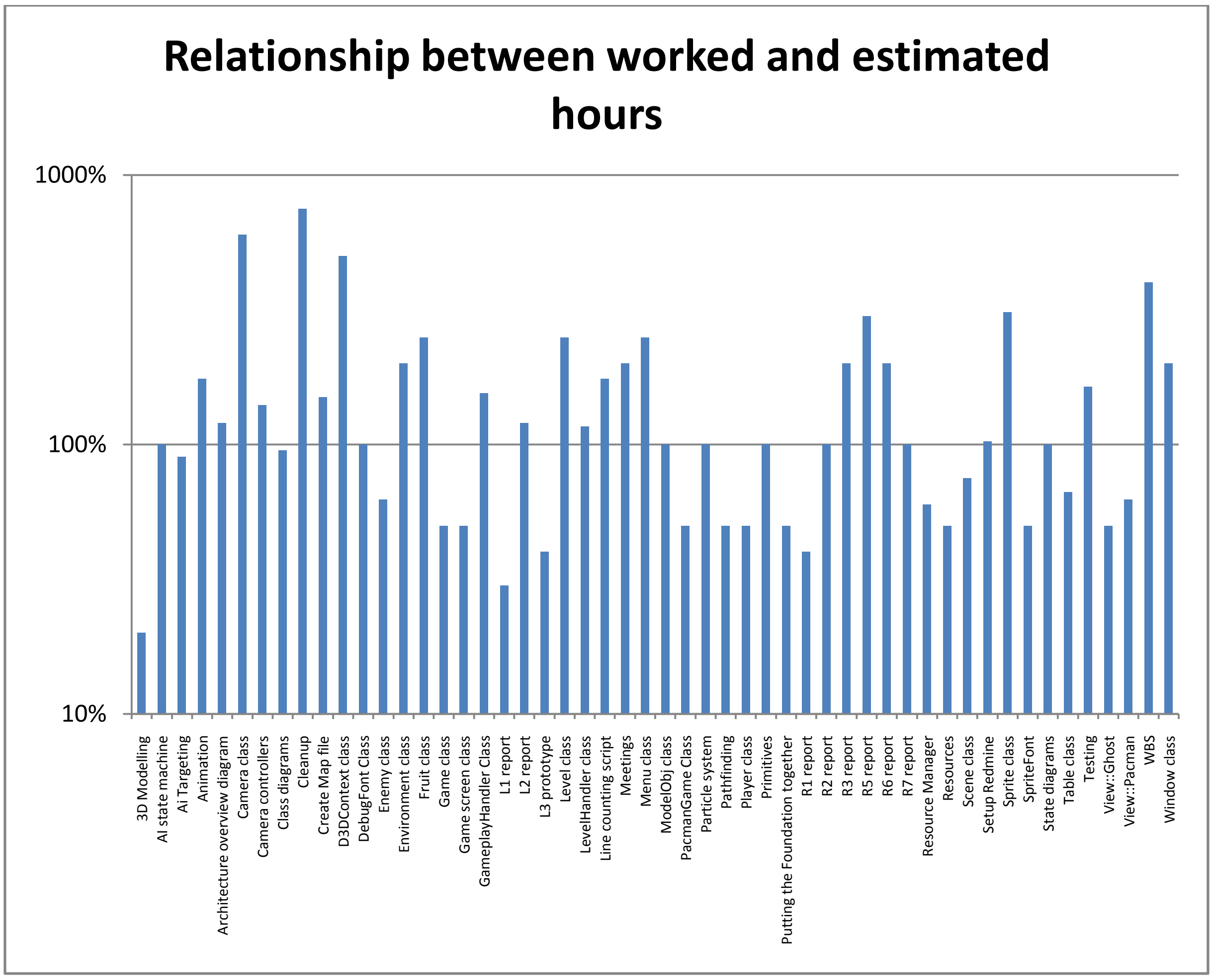
Below is a diagram that shows our project's amount of lines of code (LOC) and source lines of code (SLOC). We define a source line of code as an uncommented line with at least 4 characters (which represents the shortest statement possible in C++). Since we couldn't find a static code analysis software for C++ we had to make our own script for this, which resulted in us only having data for the last three weeks.



thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com

Estimation accuracy

Below is a diagram describing the accuracy of our time estimates. Columns represent how much time we actually worked on the components divided by the estimated time, i.e. a percentage ratio. For example, since we decided later on that we should let a technical artist do the 3D modeling, we only spent about 20% of the estimated time on that task. This diagram demonstrates clearly that time estimation was a constant issue.



The weighted average accuracy is at 116% of the estimated hours, however the values vary wildly between the highest value at 750% and the lowest value at 20%.

thsi08@student.bth.se, martin_svart@hotmail.com, lars.woxberg@gmail.com, kire10@student.bth.se, fredrik.johannesson@hotmail.com