Crush Around – Augmented Reality Game

Computer Vision and Image Processing for Mobile Platforms

|  |  |
| --- | --- |
| Tomer Cagan  cagan.tomer@post.idc.ac.il | Ziv Levy  levy.ziv@post.idc.ac.il |

School of Computer Science.

The Interdisciplinary Center, Herzliya, Israel

# Abstract

An augmented reality game in which the player interacts with the real world through her mobile device. The game models a physical representation of the scene which the user captures and as the game progress, changes the player’s view of the world using physic principals to simulate real-world-like scene transformation which is augmented on top of the current view the user sees.

# Introduction

Augmented Reality is a real time technique in which the real world environment is aggregated with additional “layers” of sensory information that are visible through the view mechanism of the user. In this project we developed a game in which the player can interact with the surrounding through the camera and screen of a mobile device, the game enhancing her perception of reality.

The game allows the player to automatically mark objects within the view and then play within that scene, interacting with the detected objects. As the game progress, the view is augmented with the results of the interaction of the user with real and virtual elements presented in the game.

The game is uses computer vision / image processing techniques for facilitating the game mechanisms:

* Object detection using canny filter and morphological operators
* Scene completion using image in-painting
* Tracking of movement using features extraction, matching, and homography.
* Output scene creation using image manipulation – transformations, blending and masking

In addition, to create real looking game experience we used physical simulation engine for movement, collision detection and other game mechanisms.

# Method

The game mechanism is a video processing loop. For each iteration, an image is captured, processed and displayed to the user. Depending on the stage of the game, different processing is done resulting in different manipulation and game state changes. The high level loop can be seen in *Figure 1: High level game loop*:

Capture Image

Input Frame

Process Image

Output Frame

Show

Figure : High level game loop

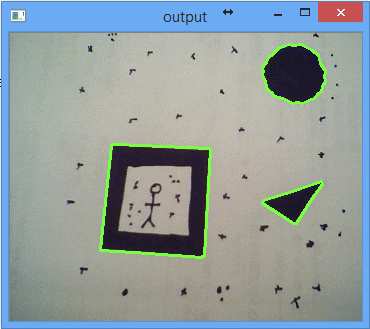
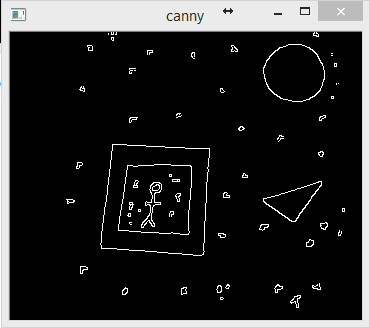
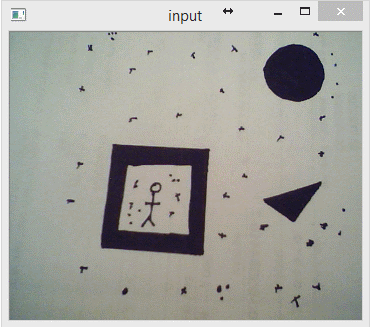
There are two processing phases – in initialization, in which object detection is done and at its end, the static resources are prepared; followed by the game loop phase in which tracking is done along with scene creation and presentation.

## Initialization Phase

In the initialization phase, the processing involved edge detection and contour extraction for regular/simple object identification followed by resource preparation in the game state initialization (called initial state or t0) as a baseline for the game-loop calculations.

### Object Detection

In each frame, a canny filter is applied to the grayscale image followed by contour extraction. The contours are then simplified – keeping the number of vertices below a threshold (executing convex hull for keeping only some vertices) and merging overlapping contours. Once the contours were processed they are overlaid over the original frame to present the objects. The process flow can be seen in the following diagram:



Canny Filter

Contours + Simplification Filter

Figure : Object Detection Flow

We have also tried to manually process the image using thresholds and morphological operators (opening followed by closing) but with lesser results.

### Game State Initialization

Once the user is satisfied with the marked objects she triggers the initialization of the game. This processed involves several sub tasks:

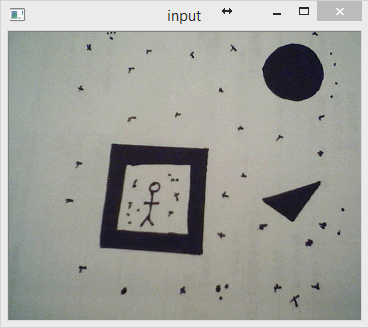
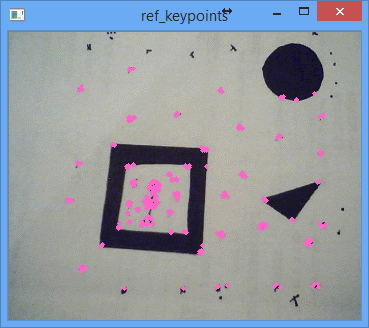
1. Setting a reference frame including feature identification and descriptors extraction which are used for tracking
2. Image in-painting inside object contours for later overlaying over regions which are “destroyed”
3. Initial scene preparation – a scene is created for tracking the game changes.
4. Physic simulation creation – creating a representation of the world.

#### Reference Features

Setting the reference frame involves detecting of features within the image and then extracting their descriptors. The descriptors are vectors, containing the relevant information of the feature as is extracted from the input image. These are later used in order to match reference descriptors to new ones in order to calculate homography.

We have experienced with several features types including Scale Invariant Feature Transform (SIFT), Speeded Up Robust Features (SURF) and Oriented FAST Rotated BRIEF (ORB). From our experience, ORB gave the best balance between execution time (for real time tracking) and tracking performance.

In ***Figure 3: Reference Features Detection*** one can view the input image overlaid with the detected reference features (pink circles). Each circle shows a key-point feature location which will later be used for motion tracking.



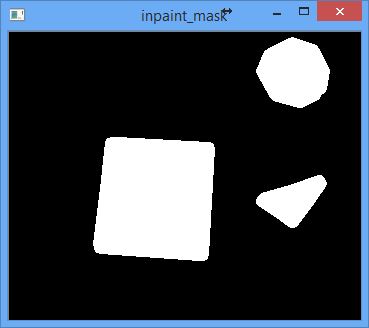
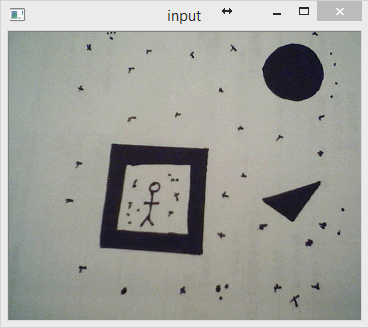
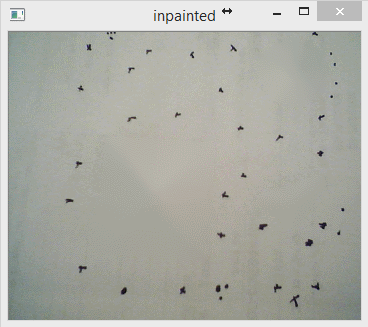
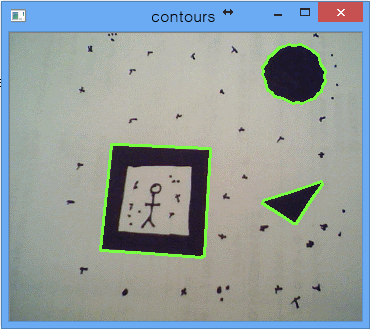
Features Detection

Descriptors Extraction

Figure : Reference Features Detection

#### In-painting

The in-painting process is depicted in *Figure 3: In-painting*. The process involves creating a binary mask image in which the internals of the identified contours are painted white indicating the areas to be in-painted by the in-paint algorithm. The algorithm then try to interpolate the content of the masked region using a configurable sliding window (threshold) around each pixel.



Create Mask

In-paint

Figure : In-painting

#### Physics and Initial Scene

As part of the initialization, an initial scene preparation takes place. The initial scene is an image that aggregates the game changes and is to be augmented onto the input from during the game. At the very beginning it is merely an image the size of screen that contains the ball.

In addition, at this stage, a physics model is also created to depict the “real” and augmented world which contains the game ball, the objects to play with and, in some playing modes, a set of fixed paddles which the player uses to keep the ball within the scene.

The initial scene and the physics world are depicted in ***Figure 5: Physics World*** and ***Figure 6: Initial Scene***. In ***Figure 5***, the green lines/shapes indicates a static object – the objects to be destroyed and the paddles; and pinkish object indicate dynamic objects – in these case, just the game ball. At the beginning of the game, there is nothing interesting in the scene as can be seen in ***Figure 6***.

|  |  |
| --- | --- |
| Figure : Physics World | Figure : Initial Scene |

## Game Play Phase

The processing during the game play phase involves the tracking algorithm, scene building and augmentation. The algorithm involves tracking of the current view port relative to the reference frame that was captured in the initialization phase, physics simulation and transformation from and to the reference features in order to keep track of game elements including the objects and ball positions, the paddles/barriers location and orientation and the destroyed scene.

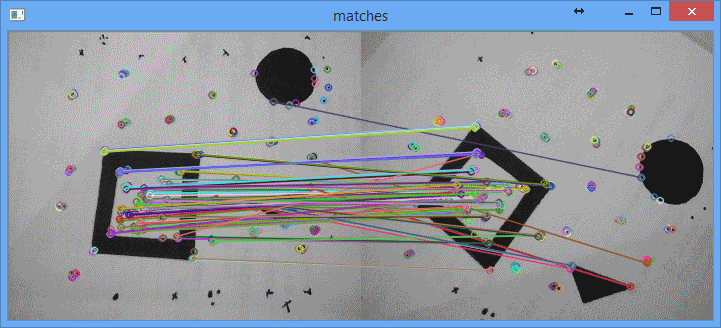
In very high level, the algorithm includes the following steps

1. Capture new image
2. Homography Calculation:
   1. Extract new features and descriptors
   2. Calculate correspondence (using RANSAC) between reference descriptors and the new descriptors
   3. Calculate homography based on matching features.
3. Physic steps simulation/updates.
4. Scene construction:
   1. Scene updates.
   2. Scene transformation.
   3. Scene augmentation.

In addition, the game also tracks user inputs (placing barriers), device orientation (moving paddles), scoring (when ball hits an object) and game state (ball is in frame).

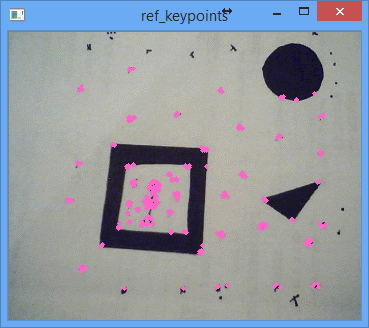
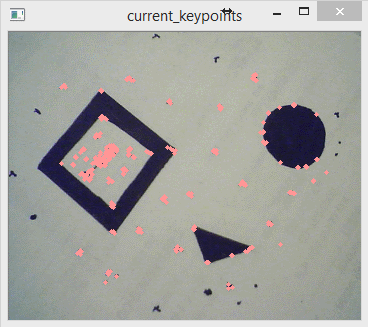
### Homography Calculation

Following the capture of a new image from the video stream, the first step of processing is to calculate the new homography. The process is depicted in ***Figure 7: Homography Calculation Algorithm*** and involves the following steps: Feature detection using ORB (1); Descriptor Extraction for the detected features (2); Finding correspondence between the current frame feature descriptors to the reference feature descriptors (3); Note that the matching is done on the descriptors which are actual vectors describing each detected feature. The matching is done using RANSAC algorithm. This may also include an optional step for only using good matches; based on the matching features compute the homography matrix (4) for the transformation between the reference frame to the current frame. The resulting homography matrix (5) is then used to calculate transformation from the original scene view point to current view point in order to build the scene and augment it correctly relative to the new view point. The inverse of the homography matrix is used for capturing user input (touches and swipes/pans) and translating them back to the reference view point for embedding in the scene as well as for positioning the paddles in the physical simulation according to the movement of the mobile device.



Feature Detection

Descriptors Extraction



Correspondence (RANSAC) Extraction

Compute Homography

Reference Descriptors

(from initialization stage)

(1)

(2)

(3)

(4)

(5)

Figure : Homography Calculation Algorithm

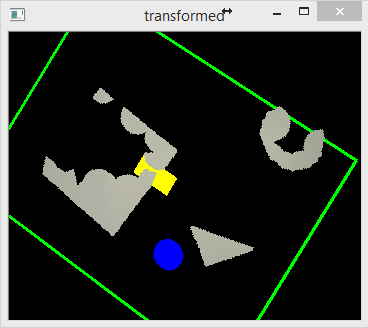
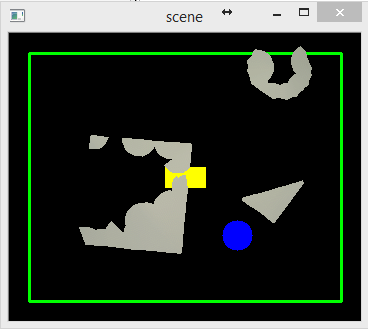
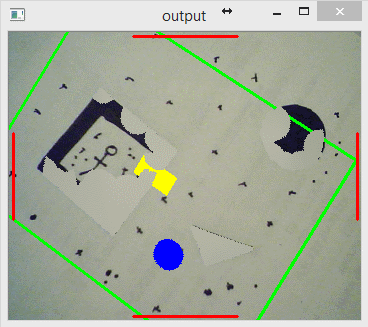
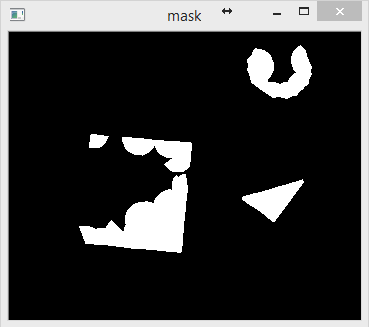
### Physics Simulation

Following the calculation of homography a few steps of physic simulation of the world take place. These calculations move objects around in the simulation and determine collisions between the objects. As a results of the calculation the physics module notify of such collisions (so the client can update state, display additional animations etc). As part of the notification, the homography matrix is used to translate the collision locations from the physical world (which are relative to the reference scene) to current view point location.

### Scene Construction and Augmentation

Once the homography between reference and current frames is known and the physical simulation is complete it is time to collect all the information, construct the current scene, transform it and augment it to the output frame to present to the user.

This process is depicted in ***Figure 8: Scene Construction and Augmentation***. The process involves the following steps: First, a mask of “destroyed” regions in created (1). The mask (2) is a binary image in which destroyed regions are painted white and the rest is left black. The mask is then used to copy the relevant regions (white) from the in-painted image (prepared during initialization) to the scene (3). At the same time, physics objects are also painted into the scene. The result scene (5) is then transformed (6) using the homography calculated before. The transformed scene (7) is augmented (8) on top the input frame to get the final results (9) which are shown to the user.



{(0.3),(2,1),(2,4),(4,3)}

Create Mask

(1)

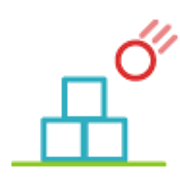
Destroyed Regions

(2)

Merge to Scene

Perspective Transform

Augmentation



In-painted image

Physics Simulation

(3)

(4)

(5)

(6)

(7)

(9)

(8)

Figure : Scene Construction and Augmentation

# Game Design

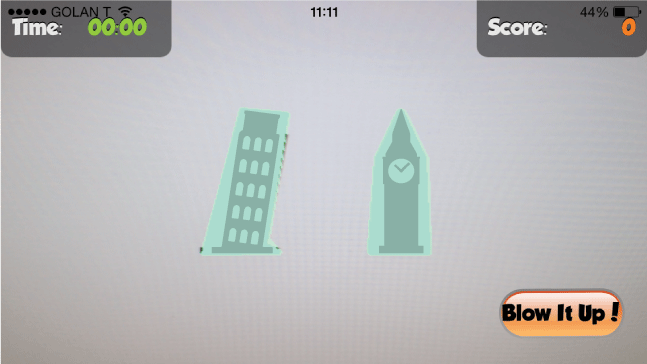
**Crush Around** is Augmented Reality game in which the player interacts with its environment – destroying it as the game progress.   
Usually, mobile games separate the player from his surrounding and his peers. Crouching on top of the mobile device, the player is isolated.   
But it doesn’t have to be that way... **Crush Around** lets the user play in its surrounding by augmenting a game layer on top of the real world which encourages him to open up to the world. By viewing the real world via camera, the player has to react to its environment. No more crouching on top of the device.



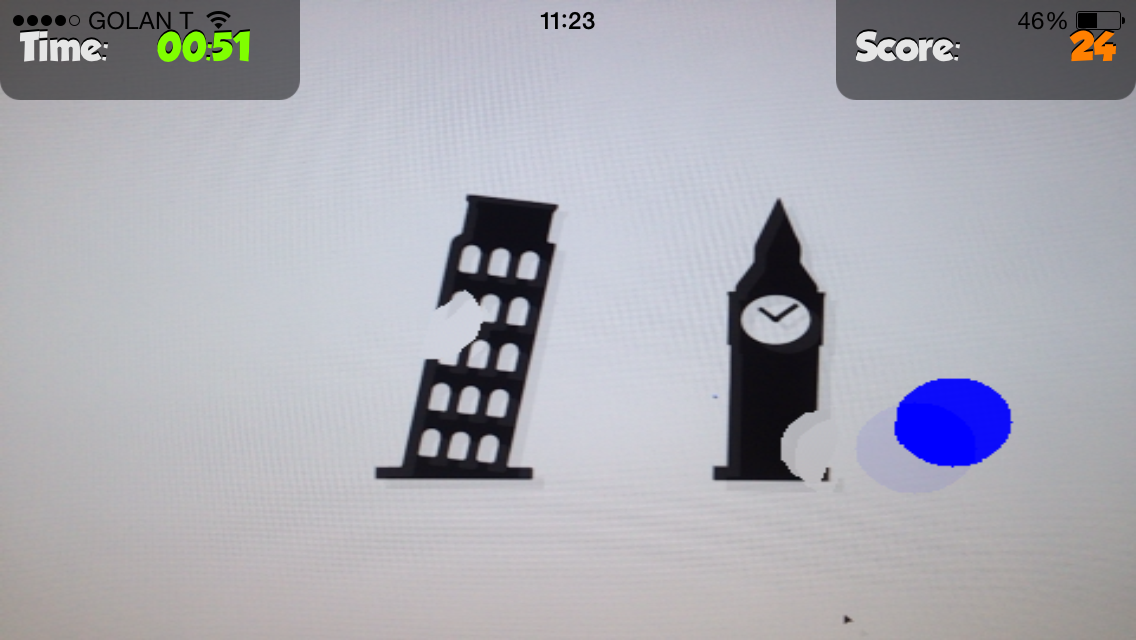
In addition, by playing in the real world, the game is meaningful not only to the player but to other people by sharing the game play epic achievements or collaborating with others in wreaking havoc.

## Game Interface

In **Crush Around** the gamer points the mobile device camera onto the scene.



The game automatically detect objects in the surrounding, transforming the scene into a game level in which the player can interact by bouncing a ball around (similar to classic “Wall Breaker” games).



The more destruction – the more points (and fun). Several game mechanisms ensure that the gamer faces dilemmas and tension, which makes the experience more rewarding.

## Code

Following you will find description of the code and sequences used in the game. The game core and the image processing classes are written in C++, uses [OpenCV](http://opencv.org/) and [Box2D](http://box2d.org/) – open source C++ libraries for Computer Vision and physic simulation so this is successfully working on multiple platforms such as iOS, Android and Win32.

### Classes

Following is a high level class diagram of the important components involved in the game

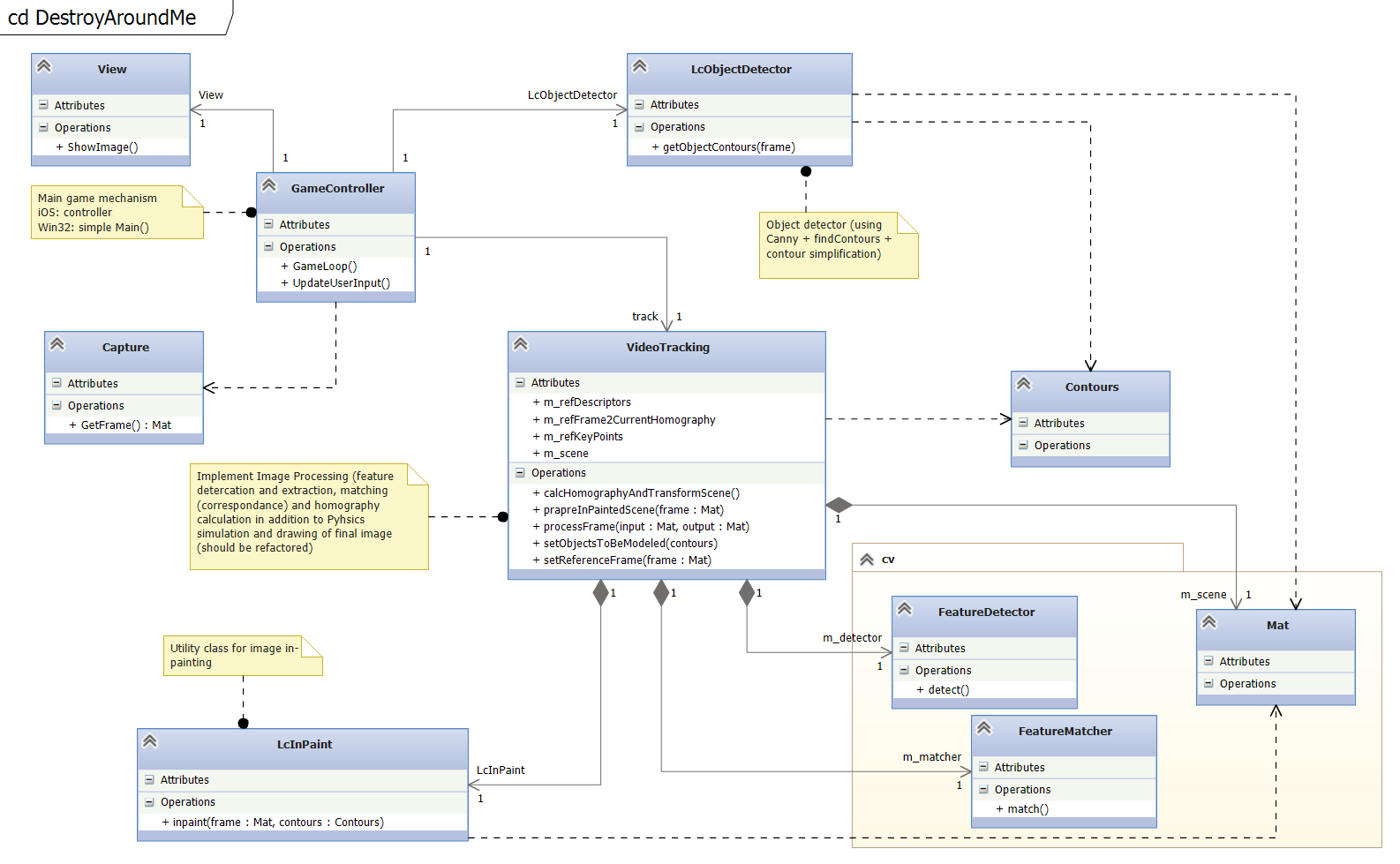


Figure : Class diagram

Following are high level descriptions of the various classes depicted in ***Figure 1: Class diagram***.

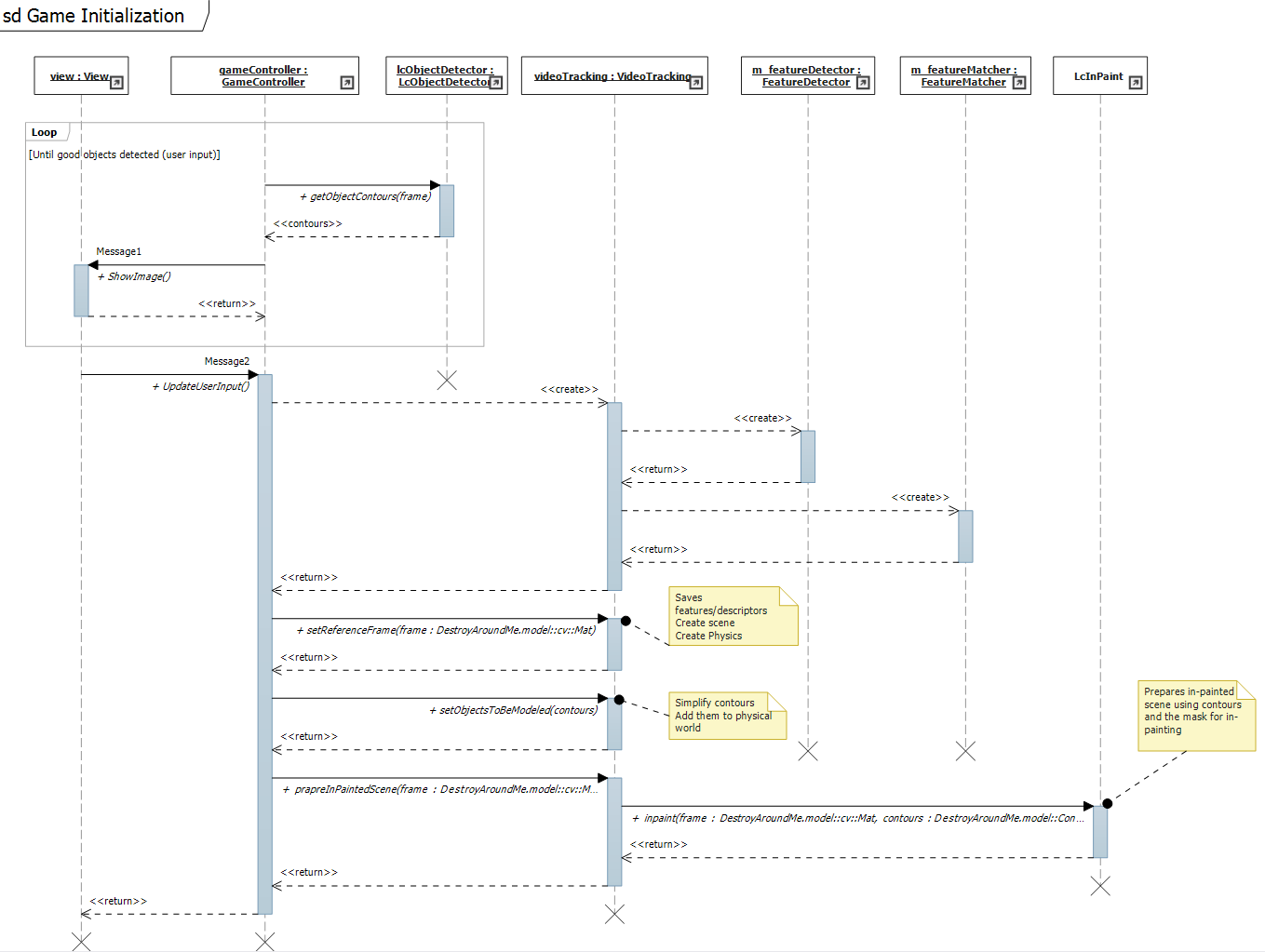
|  |  |
| --- | --- |
| Class | Description |
| GameController | The game controller running the game loop and using other classes to facilitate the actual game |
| View/Capture | External classes (platform dependent) that the GameController uses for capturing images/video and display the results. |
| LcObjectDetector | A canny bases object detector that is used to identify objects (edges) in the scene and calculate contours |
| LcInpaint | A simple wrapper class used by VideoTracking for in-painting using OpenCV inpaint() method. The class takes contours found by LcObjectDetector, offsets them and then uses them to create a mask for in-paint function. The results are saved in a member field of VideoTracking and uses to in-paint “destroyed” parts in the scene. |
| VideoTracking | The main class of the game. Used by controller to process new frame, finding features, matching them with reference features, calculates homography and transform the scene, augmenting to it the current scene state (e.g. ball, barriers (user touches) and destroyed parts). |
| Contours | Abstract notion of contours – a list of edges (either in Open CV format or other library formats (e.g. physics body fixtures). |
| Mat | Open CV matrix – used for image manipulation |
| FeatureDetector | An abstract notion of OpenCV feature detector/descriptor extractor. Currently implemented as ORB feature detector |
| FeatureMatcher | An abstract notion of OpenCV feature/descriptor matcher. Currently using FlannBasedMatcher matcher with location sensitive hash algorithm. |

### Sequences

Following are sequence diagrams describing the high level flow of the game execution

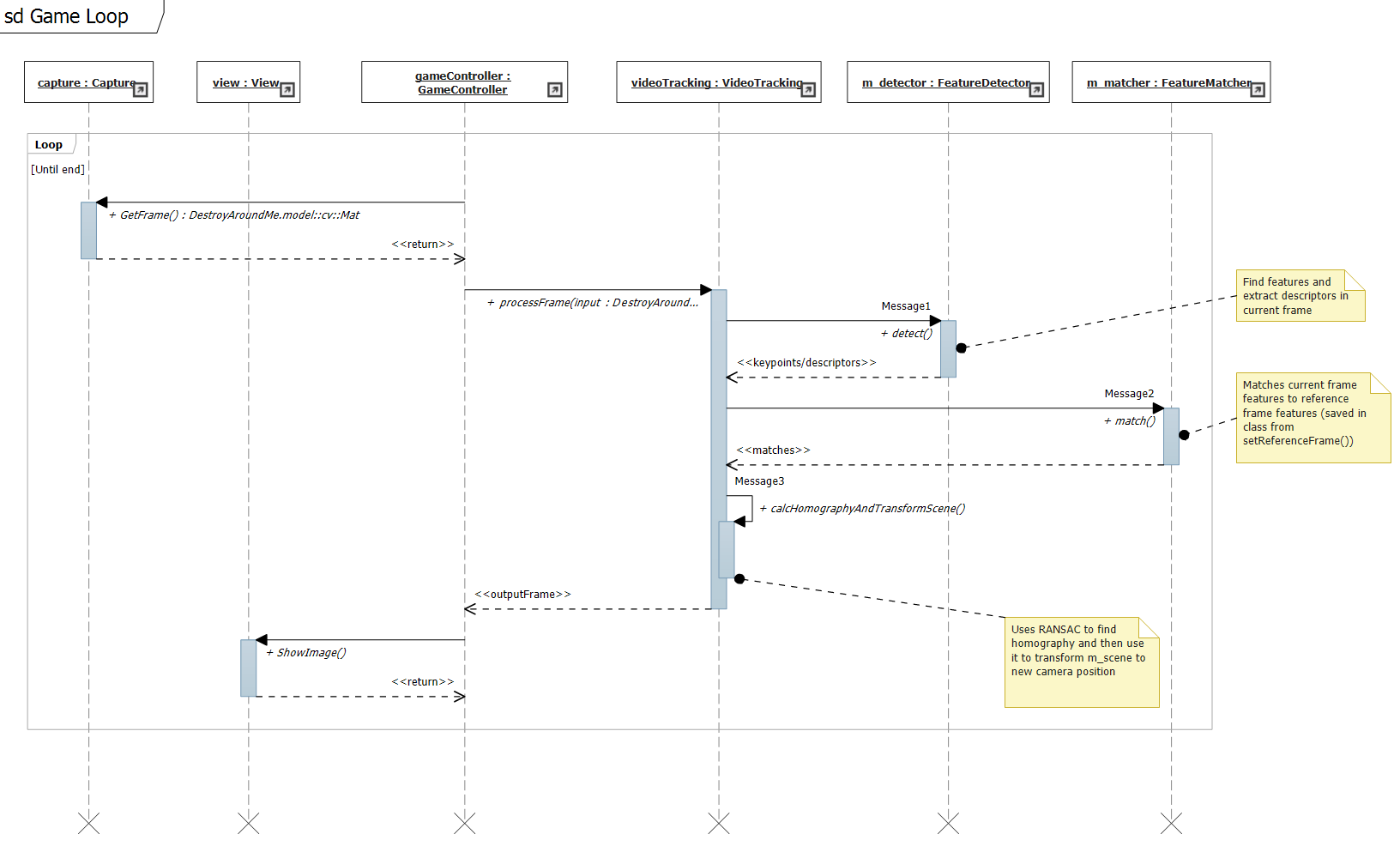
#### Game Initialization

The game initialization sequence happens at the beginning of the game. In this sequence the user (player) is displayed with a preview of the object detection results and when satisfied with the results triggers the beginning of the game which involves in the actual construction of the augmentation scene and physical model of the world in his view



#### Game Loop

The game loop is the actual game sequence in which the physical worlds in updated and the augmentation of the calculated transformed scene is shown to the user.



### Project Structure

The project is built for multi-platforms and thus have a slightly irregular structure:

|  |  |
| --- | --- |
| Global Project |  |
| Box2D | Box 2D physical engine / simulation related code. |
| Core | Core classes (described above) written in C++ |
| iOS | iOS project. Uses the core classes and iOS specific (e.g. controller, main, game mechanics) code artifacts to facilitate an iOS program |
| Poly2Tri | Not used |
| win32 | Win32 project in Visual Studio. A simple main function that facilitate the game skeleton (without fancy UI). |
|  |  |