Simulation of 2D physics of objects captured by web camera using OpenCV and Box2D

Michal Sedlák *

* *Faculty of Electrical Engineering and Information Technology, Slovak University of Technology, Ilkovičova 3, 812 19 Bratislava, Slovakia (e-mail: michal.sedlak@stuba.sk)*

Abstract

Keywords: Maximum 5 keywords.

1. INTRODUCTION

This paper describes applying of Newtonian physics to hand drawn objects recognized in image from camera. Simulation of physics is used in many modern applications. You can find in implementations used by 3D drawing and animation programs, more complex used in game engines or exact and precise simulation in CAE programs. Paper describes process of animation of hand drawn object, from a capturing phase, over recognition of the objects, interpretation objects in physical engine, to animation of such objects. This approach can be applied in education of physics at elementary schools, with interactive blackboards, or in computer games.

2. OBJECT DETECTION AND OPEN COMPUTER VISION LIBRARY

To apply a physics to hand drawn objects we need to identify and isolate objects from image. We have used a web camera as a source and Open Computer Vision library as processing tool of the images.

2.1 OpenCV

In regards the book of Bradski and Kaehler (2008) OpenCV is a library for open source programming functions for real time computer vision, with more than five hundred optimized algorithms. It can be used with C++, C and Python. We chose Python for implementation in our application.

Simple image capture is shown in Listing 1.

```
1 self.camera = cv.CaptureFromCAM(-1)
2 self.image = cv.QueryFrame(self.camera)
3 self.DetectOutline(self.image)
```

Listing 1. Query image frame from web camera

In line 1 of listing 1 we initialize our web camera. In variable camera is allocated and initialized object that can query camera for new image. Then as we see in 2 we can get the image from camera and store it in the variable named image. Capured image is shown in Figure 1.

Now when we have image data stored in the variable, we can process data to find outlines.

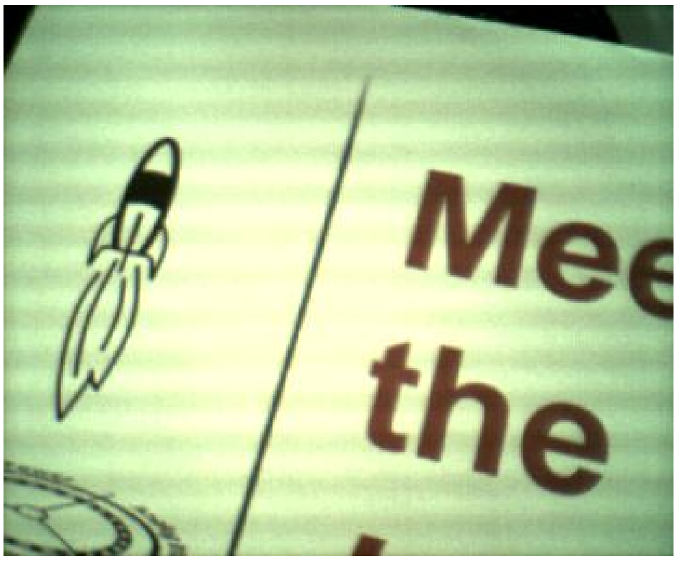
Figure 1. Image captured form camera

```
def DetectOutline(self, image):
  image_size = cv.GetSize(image)
  grayscale = cv.CreateImage(image_size, 8, 1)

  cv.CvtColor(image, grayscale, cv.CV_BGR2GRAY)
  cv.EqualizeHist(grayscale, grayscale)
  storage = cv.CreateMemStorage(0)
  cv.Threshold(grayscale, grayscale, 50, 255,
      cv.CV_THRESH_BINARY)
  self.contours = cv.FindContours(grayscale,
    cv.CreateMemStorage(),
    cv.CV_RETR_TREE,
    cv.CV_CHAIN_APPROX_SIMPLE)
  if len(self.contours) > 0:
    self.contours = cv.ApproxPoly (self.
        contours,
      storage,
      cv.CV_POLY_APPROX_DP,
      1.5,
      1)
  return self.contours
```

Listing 2. Outline detection

In function in Listing 2 is shown how to find outlines of objects in image. We convert image to gray scale as seen on line 3.

Then we run histogram equalization (line: 5). Equalization makes objects better visible and gives better output for thresholding (line: 7) which makes black and with image prepared for outline detection (line: 8). Output of thresholding is shown in (Figure 3).

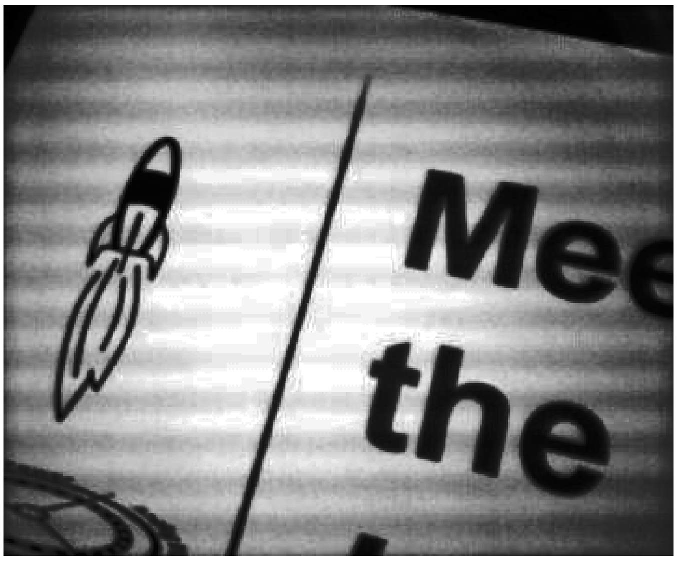

Figure 2. Image after histogram equalization

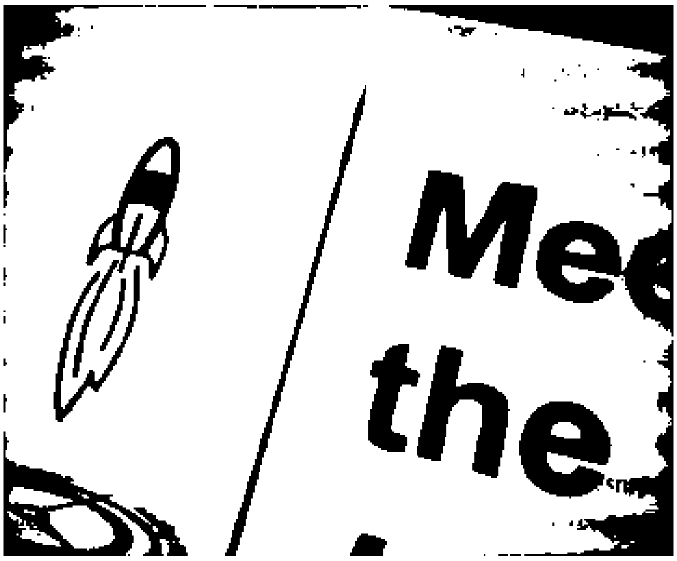

Figure 3. Image after thresholding

After outline detection we have tree of contours stored in the variable self.contours. These trees are iterable objects sorted from outer to inner outline connected by property h_next and v_next that we will describe in paragraph about creation of objects from outlines.

Contour can be very complicated and consist of thousands of points, which could cause too complicated objects. It is time demanding to simulate complicated objects, that is why we use polynomial approximation of the contour points. (line: 13). Visualusation of outlines is shown in Figure: 4.

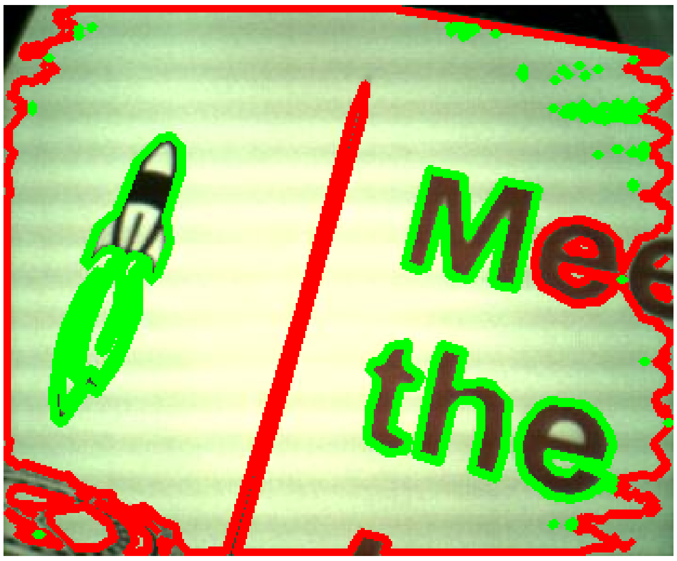

Figure 4. Visualisation of contours

Now we have all outlines stored in the outline tree structure, so we can create objects and apply a physics.

3. PHYSICS SIMULATION IN BOX2D

There is lot of physics engines that can be used for simulation of physics. Because we wanted to simulate physics only in 2D we could, code our own implementation of physics, or use one of commercial or open source engines. We chose Box2D[Thorn (2010)], which is open source 2D physiscs engine with possibility to simulate rigid body objects and their collisions.

3.1 World

To create physics simulation we need to create world. World is object that manages memory, objects and simulation. Creation of world is shown in Listing 3:

```
1 self.worldAABB=box2d.b2AABB()
2 self.worldAABB.lowerBound = (-100.0, -100.0)
3 self.worldAABB.upperBound = ( 600.0, 600.0)
4 gravity = (0.0, -10.0)
5
6 doSleep = True
7 self.world = box2d.b2World(self.worldAABB,
      gravity, doSleep)
```

Listing 3. Creation of Box2D world

First we have to create boundaries of the world. We define them as vectors from bottom left (line: 2) to top right (line: 3). Objects have to be inside the boundaries, when an object touch the boundary it gets stuck. Then we define gravity vector (line: 4). The last thing before creation of the world we allow objects to sleep(line: 6). Object that are not moving fall asleep, then are ignored by the engine. Last line of Listing 3 creates the world.

World is created and we are ready to create objects from outlines.

3.2 Objects

Every object that is simulated in Box2D consists of body and shapes. Our objects are described by the contour tree. To create objects we need to iterate contour tree, find contours that belongs to together and create objects for these contours.

Contour tree Contour tree is object in which are stored points of each contour. Contours are connected by functions returning reference to other contours with h_next() and v_next(). h_next() is referencing to deeper contour, and v_next is referencing to another object contour. To iterate over all contours we have created recursive function shown in Listing: 4.

```
1  def CreateObjectsFromCountours(self, cont, h=0,
       v=0):
2    if v>0:
3      density = 10.0
4    else:
5      density = 0
6    if len(cont)>8:
7      self.CreateObject(cont,h,v)
8
9    if cont.v_next():
10     v += 1
11     self.CreateObjectsFromCountours(cont.v_next
           (),h,v)
12     v -= 1
13
14   if cont.h_next():
15     h += 1
16     self.CreateObjectsFromCountours(cont.h_next
           (),h,v)
```

Listing 4. Function to iterate through contour tree

We iterate through contour tree. First level is outer contour of the image (line: 2), because we do not want the outer contour to move, we set it as static byt setting density to 0 (line: 5). Every other contour is dynamic body with density set to 10 (line: 3).

When we know what type of object we will create, we can create bodies and shapes for our contours.

Bodies, shapes and collisions Bodies are backbone used by shapes. One body can contain more shapes, but one shape could be attached to only one body. Box2d is rigid body physics engine, that mean that shapes attached to body can not move against other, or body. Body have position and velocity. Forces, torques and impulses can be applied to body [Catto (2010)]. Bodies just hold the shapes. Shapes are elements that collide together.

Listings 5,6,7 shows how to create object from outer contour:

```
1  def CreateObject(self, cont, h,v):
2    contM = []
3    for point in cont:
4      x = point[0]/30.0
5      y = point[1]/30.0
6      contM.append((x,y))
7
8    bd=box2d.b2BodyDef()
9    bd.position = ( 0.0, 0.0 )
10
11   edgeDef=box2d.b2EdgeChainDef()
12   edgeDef.setVertices(contM)
13
14   if v==0:
15     body = self.world.CreateBody(bd)
16     try:
17       self.contourBodies.append(body)
18     except:
19       self.contourBodies = [body]
20       body.CreateShape(edgeDef)
```

Listing 5. Creation of object from outer contour

Image size is measured in pixels and Box2D units are kilograms, meters, and seconds (KMS) we should scale images coordinates to fit in 0.1m to 10m. In that scale is performance of Box2D the best. We are doing it by dividing of value of pixel coordinates by 30.0 (line: 4)

Then we create a body definition that will represent our contour(line: 8) and set up it initial possition in next line.

After that we create shape of body as chain of edges (line: 11) and assign the array of vertices to it (line: 12). Edges are special type of shapes that have no mass, is represented as line betven vertices that collide with other non-edge objects. Edges are easy to create because they do not have to be concave unlike polygons.

At last we attach this shape to created body 20. Because Box2D does not keep track about body definitions, we have to store bodies in to array for later use19.

Listing of the function CreateObject() continous in Listing 6. This part of function creates dynamic objects inside the outer contour. In this part we prepare list for bodies of objects, so we can modify objects that are already created or objects that we want append new shapes.

```
21     try:
22       body = self.objectBodies[h]
23     except:
24       body = self.world.CreateBody(bd)
25       self.objectBodies[h] = body
```

Listing 6. Creation of objects

3.3 Tesselation

Box2D supports only collisions between convex objects and contours of objects captured by camera are not convex we have to break outlines to convex polygons. There is more ways how to break concave objects. We chose the 2D constrained Delaunay triangulation algorithm implemented by poly2tri Python library[Rognant et al. (1999)]. Function CreateObject() continous in Listing 7

```
26     polyline = []
27     for (x,y) in cont:
28       polyline.append(p2t.Point(x,y))
29     cdt = p2t.CDT(polyline)
30     triangles = cdt.triangulate()
31     for t in triangles:
32       x1 = t.a.x/30.0
33       y1 = t.a.y/30.0
34       x2 = t.b.x/30.0
35       y2 = t.b.y/30.0
36       x3 = t.c.x/30.0
37       y3 = t.c.y/30.0
```

```
38          if math.hypot(x2-x1,y2-y1)<0.1:
39            x2 = x2 + math.copysign(0.1, x2-x1)
40            y2 = y2 + math.copysign(0.1, y2-y1)
41          if math.hypot(x3-x2,y3-y2)<0.1:
42            x3 = x3 + math.copysign(0.1, x3-x2)
43            y3 = y3 + math.copysign(0.1, y3-y2)
44          if math.hypot(x1-x3,y1-y3)<0.1:
45            x1 = x1 + math.copysign(0.1, x1-x3)
46            y1 = y1 + math.copysign(0.1, y1-y3)
47          poly=box2d.b2PolygonDef()
48          poly.setVertices(((x1, y1), (x2, y2), (x3
              , y3)))
49          poly.density = 1.0
50          poly.restitution = 0.0
51          poly.friction = 0.0
52          body.CreateShape(poly)
53          body.SetMassFromShapes()
```

Listing 7. Creation of objects

The creation of objects continues with tesselation. We need to assigne vertices to structure that could be understood by poly2tri library (line: 28) and we initialize the CDT object (line: 29). In next line we call function that will create triangles from the verteces assigned before. These triangles are in image pixel coordinates, so we need to scale them at first (lines: 31-37). Now when we have triangles scaled we need to scale the triangles that are too small to triangles with size at least 0.1m because of speed optimalization, this is done in lines: 38-46). Now we have set of tirangular shapes that could be attached to body (line: **??**). Because these objects are compound objects, we need to set the mass center and ammount to this body. We can let Box2D set this properties based on shape information with body function SetMassFromShapes() (line: 53). Visualisation of objects is in Figure reffig:objects

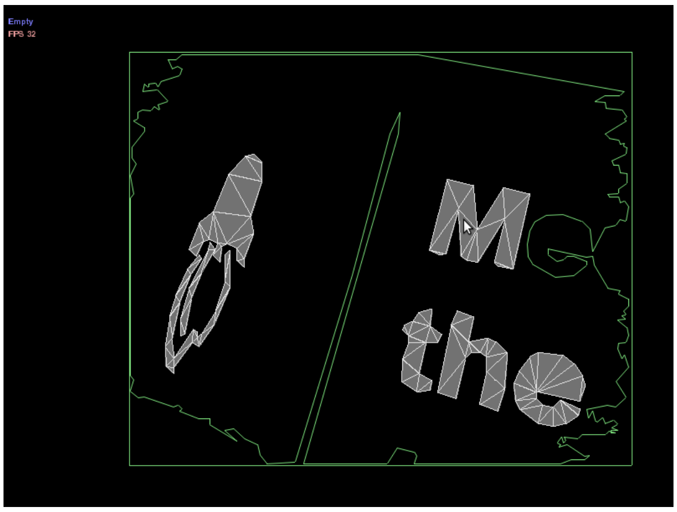

Figure 5. Visualisation of objects after triangulation

After this we can add other objects and start simulation by function Step(). After few second of simulation are all objects on the bottom of the screen like in Figure 6.

4. FUTURE WORK

Identifikacia objektov Sledovanie objektov a morfing Interakcia hybucich sa objektov zachytenych kamerov s Box2D reprezentaciou

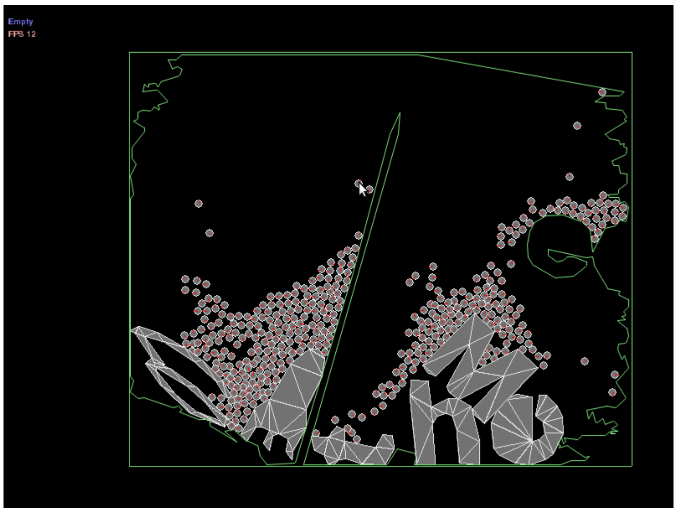

Figure 6. Visualisation of simulation

ACKNOWLEDGMENTS

The work was supported by a grant (No. NIL-I-007-d) from Iceland, Liechtenstein and Norway through the EEA Financial Mechanism and the Norwegian Financial Mechanism. This project is also co-financed from the state budget of the Slovak Republic.

REFERENCES

Bradski, G. and Kaehler, A. (2008). *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly, Cambridge, MA.

Catto, E. (2010). Box2D v2.0.1 User Manual. `http://code.google.com/p/pybox2d/downloads/detail?name=2.0.2%20documentation%20from%20wiki%20archive.zip`. [Online; accessed 20-January-2011].

Rognant, L., Chassery, J., Goze, S., and Planes, J. (1999). The delaunay constrained triangulation: the delaunay stable algorithms. In *Information Visualization, 1999. Proceedings. 1999 IEEE International Conference on*, 147 –152. doi:10.1109/IV.1999.781551.

Thorn, A. (2010). *Game Engine Design and Implementation.* Jones & Bartlett Publishers, Cambridge, MA.