Visual Interfaces to Computers

Final Project

Report

Mauricio Castaneda

mc3683

mc3583@columbia.edu

Columbia University

Spring 2013

Conversion of Static Images to Dynamic Scenarios

Motivation

There are certain images for which it is very easy to predict, as a human being, what is about to happen. It is easy for us because humans have the capacity to identify the scenario, and using our previous knowledge, previous experiences and known properties such as gravity, we are able to predict what will happen next. Consider the following image on the right:

It is clear (for most human beings at least!) that this apple will hit Newton's head, because we understand the fact that gravity has an effect on objects. However, for computers this is not a simple task. A computer cannot understand the context of this situation, and even though there are simulation environments, they can't identify and predict what will happen next from the context of a simple image, even as basic as the one shown.

Previous Work and References:

There has been work done to address this problem. For example, this patent describes a system that simulates the weather, based on a video film. The system detects falling objects, and tries to simulate an environment that meets these conditions: http://www.google.com/patents/US20100153078. However, this runs on footage, and it simulates small particles such as rain and snow.

There was a paper published at a Computer Science conference in Slovakia where gravity is simulated from images. However, in this paper, the shape and color of the objects are not taken into account. The text, or shapes from the image are converted into rectangles and then they are "shattered" and fall against the lower part of the image. The "ink particles" are affected by gravity. In this case, there is no interaction based on different kinds of properties.

Source: http://www.kirp.chtf.stuba.sk/pc11/data/papers/053.pdf

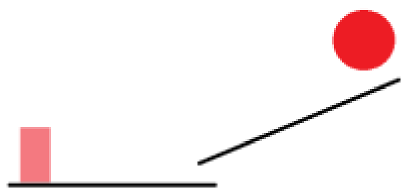
Another project shows a system where a checkerboard is used to rotate and control a "virtual box" that can be loaded with "virtual balls" that are affected by gravity, and interact amongst each other. The user can then manipulate the checkerboard, moving the virtual box, and moving the balls contained in this box.

Source: http://srinivaseducation2.doomby.com/videos/cet-and-puc/cet-and-puc-physics/computer-vision-and-physics-simulation-game.html

There is a tool that has a physics simulation engine that can be combined with other platforms such as openCV. Information about a simulation tool can be found at this website. The developing community is large and there are many forums that respond to problems other users are encountering. The website is: http://unity3d.com/

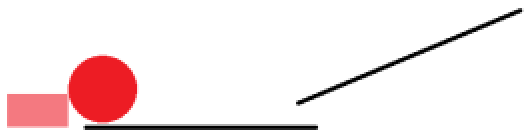
Description and Limits

The idea of this project is to receive a static image (jpeg or ppm) with blobs of different colors and sizes and through visual information extraction, have these blobs interact in a dynamic environment. In order to achieve this, certain properties of these "blobs" will correspond to certain physical properties and then this information will be loaded into a dynamic environment where it is possible to see the objects interact with each other. For example: given a certain image, all the objects that are black must be static, however, the objects that are red, are solid and dynamic. To take this even further, the deepness of the color red will be used as a measure of how "massive" or "heavy" the object is. Since it is very difficult to take into account texture and perspective in order to generate a simulation environment, a 2D, simple, "flat" image will be used as an input. To illustrate this example, consider the following input image:



After the information is extracted, and the simulation is executed (in a real-time visual environment) the results can be described in the following timeline: 1. The red ball will drop onto the static downwards ramp. 2. The ball will roll down the ramp, onto the lower static black plane. 3. The ball will continue rolling (due to its momentum) until 4. It hits the pink block. 5. Since the "pink block is less dense than the red ball, it will tip it over, and both objects will fall.

The final configuration would be as shown in the following image:



The idea is for the whole process to be animated automatically, and the simulation to run smoothly. This does not imply that the reverse process will occur as well, and that a final image will be produced from the simulation (the conversion only works one way).

Additionally, other information about the color will be used in the simulation as additional information. For this project the Hue is used to determine the "nature" of the object. This means that if an object is red, it will be an incompressible solid (not bouncy, some friction). If an object is coded as blue, the object will be coded as a somewhat compressible solid (bouncy effect), and finally, if the object is black, it will be coded as an incompressible solid unaffected by gravity (used as platforms or “walls” in the simulation).

Procedure

1. Blob Detection

The first step in this project is to extract the different shapes. Initially, the system will recognize lines (thin rectangles), circles, and rectangles. These shapes must be recognized from the image, and its relative location to the other images must be registered.

For the detection of circles, the Hough Transform method was used. This method helps detect circles in an accurate manner using varying parameters. Using this method, the application is able to extract the radius of the circle and its central coordinate.

In order of this method to work correctly, first, a black and white copy of the original image must be made. This will ensure that circles of all colors will be detected. Then, a Gaussian blur is applied to remove noise that the image might have. Now, the image is ready for the Hough Circles method. Once this is done, the circles are extracted and are added to a list, ready to be instantiated into physical objects. Finally, to reduce some of the error that occurs when using the Hough Circles method, all circles that have a white center (not circles of our interest and possible bugs) are pre-filtered, and not added to this list. The code can be seen in the following lines:

```
void findCircles (IplImage imgSrc)
{
    using (IplImage imgGray = new IplImage(imgSrc.Size, BitDepth.U8, 1))
    using (IplImage imgHough = imgSrc.Clone()) {
        using (CvMemStorage storage = new CvMemStorage()) {

            //Prepare image for hough processing
            //Pre-Processing
            Cv.CvtColor (imgSrc, imgGray, ColorConversion.BgrToGray);
            Cv.Smooth (imgGray, imgGray, SmoothType.Gaussian, 9);

            //Do HoughCircles
            CvSeq<CvCircleSegment> seq = imgGray.HoughCircles (storage, HoughCirclesMethod.Gradient,
                                                    2, imgGray.Height / 3, 200, 55, 10, 100);
            for (int i = 0; i < seq.Total; i++) {
                CvCircleSegment item = (CvCircleSegment)seq [i];

                //Add circles that dont have a white center (not real circles!!)
                //This is sort of a hack, additional level of protection...
                CvColor color = imgSrc[(int)item.Center.Y, (int)item.Center.X];
                if(!(color.R > 230 && color.G > 230 && color.B > 230))
                    circles.Add(item);
            }
```

The line and box detection was done by identifying contours of the objects in the image, approximating (smoothing) these contours, and finally counting the sides. More specifically, the procedure is as follows: First, there is a reduction of noise by downs-scaling and up-scaling the image. Then, a black and white copy of the image is generated. Once this black and white image is generated, a canny version of the image is created. This creates an image with just the edges in the image, which will be very helpful for the detection of contours. The next step is the actual extraction of the contours using an opencv function that has various threshold parameters including the thickness of these contours, the space between them, and other threshold values. Once the contours are obtained, an approximate polygon is generated (also using an opencv function). With this approximate polygon it is possible to differentiate lines and boxes.

In the case of a rectangle, the polygon has exactly 4 vertices and they are connected in a convex polygon. Also, all the angles of this shape are close to 90 degrees. For the case of a line, the approximate polygon is so small, that it only has two vertices, and thus a line can be extracted from these two points. The information extracted from this procedure is stored in arrays that will later be used to create an appropriate representation that will be used in the simulation.

2. Model Translation

Now that the blobs are identified, an appropriate representation in the simulation environment must be found. These new shapes that are to be created must have proportional sizes and similar shapes to the ones found in the image. Specifically, a translation to a 3D environment was used. This approach was selected because there are many physics simulation tools available for 3D environments that have a good and consistent behavior out of the box. However, assumptions had to be made. In order to restrict the motion to two dimensions, all generated objects have a z-plane restriction, and cannot move through this axis. The next issue encountered is the fact that the images have no absolute coordinates, nor do they have any measure of distance. In order to address this issue, a 10X10 meter “canvas” was chosen for all the simulations. Therefore, all the objects in the image were scaled accordingly. This means, if a box is half as wide and half as tall in the image canvas, this object was assigned a 5 meter width and 5 meter length. Also, all the images were that were used have to have a fixed size of 300x300 pixels.

For the circles, an appropriate 3d equivalent was used. Since the radius of each circle was known, these objects were extended to a 3d version, and thus, spheres were generated. In the case of rectangles, boxes were used, and the depths of these boxes correspond to a fixed value. For the lines, a simple plane was not used, since this is hard to visualize. Instead, thin boxes of fixed thickness and depth were used.

Other assumptions include the weight and density of the objects. Since we cannot physically weigh the objects, or manipulate them directly, assumptions about a maximum mass, friction, and “bounciness” factor were selected arbitrarily. In order for the elements of the simulation to interact in a meaningful manner, a maximum mass of 100kg was selected for the heaviest objects. The bounciness factor, and the friction involved, as well as the mass, depend on the color of the object

3. Color Detection and Translation

The next step consists of identifying the color of each of their shapes. Three colors were coded in this application. The color black was used to represent solid and static objects that are unaffected by gravity, red to represent solid, incompressible, and non-bouncy objects, and blue to represent solid, bouncy objects. Also, depending on the intensity of each of these colors, a measure of how much of this property the given object has (i.e. bluer objects will be bouncier). This will allow for a greater interactivity within the objects. More specifically, the corresponding color of an object in HSV was taken into account to code for 3 different properties. This way, it is easier to extract the required information, and visualize the differences between the objects.

The “bare” color of an object (i.e hue, as mention before) was used to codify for the “type” of object that is to be generated. The saturation was used to represent the mass of the object. This means that the more saturated the color of an object is, the greater it’s mass. However, there is a maximum mass (which was defined to be at 100kg) for a completely saturated color, and black objects do not follow this restriction (they are unmovable). Finally, the value of the color was taken into account to code for the amount of friction an object has, and it also codes for the “bounciness” in the case of a blue object. With these parameters we have an interesting simulation environment.

4. Simulation

Once all the values are set, and an appropriate model is generated for these images, a simulation of these objects is generated. This simulation runs smoothly using the Unity3d physics engine. The objects interact with each other, taking into account their physical properties. Gravity and the other physical properties are taken into account. This is done by using and adjusting some of the built in functionally of this engine. Specific objects are created, and each one of them is attached a collider, and other components that allow them to have a simulated physical interaction. Also, the material of each object’s collider is modified to meet its specific conditions. The simulation runs under the conditions previously defined, and generates an accurate model.

Results

In order to test the correctness of the application, seven test images were used. These images can be viewed in the corresponding source code folder. These images consist of different scenarios of these three different objects interacting with each other.

Img1 and img1a: Both these images contain similar scenarios which consist of a circle about to fall into a ramp that leads to a rectangle. In the first image, the ball is much heavier than the rectangle, and this rectangle has a small coefficient of friction. As is expected, the ball knocks the cube over. In the second image, the rectangle now has more mass than the circle, and it also has a higher friction coefficient. In this case, the ball cannot knock down the cube.

Img2: This image consists of two blue balls (with a high “bounciness” coefficient) about to fall on a blue surface. As is expected, the balls bounce for a while, and then come to a full stop.

Img3, img3a, img3b, img3c: These scenarios are somewhat similar but combine different elements. The first one is a simple red ball falling down a ramp until it is stopped by a black line. The second one is of a similar ramp, but the ball then bounces of a blue surface. The last two are very similar, a ball falling onto a blue object that pushes the ball up the ramp. However, at this point, there is an inconsistency in the physical model used. The ball used in the first case has a greater mass than the one in the second case, yet since they both hit a blue "trampoline" like object, they would be expected to bounce the same distance back. However, this is not the case, and the lighter object gets pushed back further.

Img4: This image is used to show that the circle and rectangle detection doesn't always work. In this case, a sphere is generated where there clearly isn't one, and the simulation acts in a manner that was very difficult to predict, due to the nature of the circle, rectangle, and line detection.

Conclusions

There were interesting results from the development and testing of this application. Specifically, circles, rectangles and boxes were detected accurately in most scenarios. However, it is difficult to create a system that will identify all the shapes correctly. This error is further magnified when trying to build something on top of this information. The fact that there is an initial image feature detection that later is used as an input for a physics simulation just amplifies errors and causes unpredicted behavior of the system. However, it is worth noting that there is much information that can be extracted from simple images to create a context that a machine can understand.

Future Work

There are many applications for a system of this nature, and many possible expansions as well. First, it would be of great use to extend the functionality of this system to not only include basic "flat" images, but any kind of image, with humans and other shapes, and have a machine try to predict what will happen. These could include waves on a beach, planes about to land, baseball players about to hit a ball, amongst others.

Another interesting application for this type of system, that might be in a closer future include classroom learning. A system like this could be integrated with a system like the Columbia video network, where professors use a virtual interface to generate images. With the help of a system like this, simple drawings made by a physics professor could be simulated to view the results of such a setup. In the future, it could be possible to have the values of mass, friction constants, spring constants, and other physical properties mapped not as colors, but as side notes close to the corresponding structure. This could also be used to visually simulate circuits and other more complex structures directly from the classrooms board.

Tools

- Windows 7
- Opencvsharp (C# wrapper for opencv)
- Unity3D – Free Version

References and Credits

All code written by Mauricio Castaneda

Three references mentioned in the “previous work” section. Out of these, there is only one published article, and the rest are short descriptions and a video of how the application works.