Scene Geometry Generation for Augmented Reality using Markers for Interactive Experiences

Thaminda Edirisooriya
Stanford University
450 Serra Mall, Stanford, CA 94305
tediris@stanford.edu

Abstract

I present an application of Computer Vision and graphics techniques used in tandem to generate an interactive experience where users unfamiliar with augmented reality applications can see and interact with a real-time simulation with simple physics. The system depends on the use of augmented reality markers to localize the world, and to allow for user interaction - moving the markers in space moves objects in the simulation, as other objects remain static relative to the motion of some markers. Objects can bump one another and simple rigid body physics will govern the interactions. Vision algorithms such as Canny edge detection, Homography transformations, and Pose estimation are all used in conjunction with OpenGL 3 drawing pipelines and the react3D physics engine to deliver an experience that delivers richer interactions to the user. The project extends existing literature by combining methods that have been developed and applying them in new ways that create interesting experiences for users, especially those unfamiliar with modern augmented reality methods and possibilities. Most of the work presented here pertains to combining subsytems developed in previous works in ways that allow for real-time, augmented reality simulations of rigidbodies controlled by markers positioned by users.

1. Introduction

Many augmented reality applications allow users to project 3d images and models onto markers and surfaces in images taken of the real world. However, they often stop there, with few capabilities beyond that. Interactions with the applications are limited to moving the camera around, switching the models, and simple interactions between models.

I sought to create a more interesting, tangible application that leverages augmented reality methods, as well as techniques from computer graphics and game development. The application provides an experience that can be manipulated in real time by anyone without understanding of the underlying system.

Before anything, the camera is calibrated, and the internal camera parameters stored for later use.The application pipeline works as follows: first, the camera images are captured and processed, and the Canny edge detection algorithm is run [2]. Second, edges are filtered to find the outer borders of the augmented reality markers. Third, the markers are identified, and their pose and position computed. Fourth, the pose and position of each detected marker is calculated and fed into the graphics stage. Next, the graphics stage uses the pose and position of the markers to localize the virtual world and position elements of the simulation in the world. Finally, all the information is used in the physics simulation to update the physics of the world on each timestep, and the final results of this are sent to the OpenGL rendering engine to provide the visualization.

The vision aspects of the algorithm were initially done in python using OpenCV for functions such as Canny edge detection, and homography computations, but the online nature of the application required faster computation, and we instead used a C++ implementation of the marker detection. The marker detection module used is part of OpenCV. The bulk of the application is the rendering engine which was implemented using raw OpenGL, and tying in the data from the vision pipeline and combining it with a simple physics engine, as well as building a ”game loop” that could keep the simulation running at a constant time, regardless of CPU usage.

2. Previous Work

2.1. Existing Work

The main inspiration for this project, as well as one the enablers for this work and much of the existing work in augmented reality, comes from [3]. The paper illustrates a solid ground work for detecting augmented reality markers and solving for their position and pose. However, it ends there without presenting many of the interesting applications of the methods it develops.

Other existing works [5] talk about non-marker related methods for solving for world positions for augmented reality applications, but again do not delve into more interactive experiences involving high precision markers and simulations of worlds.

2.2. Our Methods

The methods we present here take advantage of the groundwork and ideas put forth in the aforementioned works and applies them in a novel way - in the form of an application which users can interact with to create tangible differences in the simulated world. We present an extended graphics pipeline that uses OpenGL 3 shaders to quickly make use of the vision information to render the visualization, and we construct an underlying physics model that uses information from the vision pipeline to position objects in the simulation relative to each other.

The result is a system that allows users to physically move markers around, even bump them into each other, and see the effects in a gravity-less physics world of simple box shapes bouncing off of each other. It is expandable to other even more interesting applications, such as an augmented reality pong game, due to all the required systems (vision, graphics, and physics) being implemented and working in tandem.

3. Technical Solution

3.1. Summary

As discussed earlier, the pipeline consists of a vision component, a graphics component, and a physics component. The vision component exists to detect the augmented reality marker, and compute its pose by setting up a linear system and solving for it's nullspace. Afterwards, the information is fed to the simulation engine, where it is used to update physical locations for objects in the scene. Finally, the outputs of both the vision pipeline and the physics simulation are output to the graphics pipeline, where OpenGL shaders are used to render the supplied 3D models onto the scene either on top of markers present in the scene, or relative to a base marker in the scene that describes the origin of the simulation world.

3.2. Camera Calibration

The camera is calibrated using a variation of the OpenCV findChessBoard corners that makes use of the augmented reality markers to allow for occlusion of the board [1]. Figure 1 below shows the image used for calibration. Figure 2 shows a snapshot of the calibration process. A standard method of solving for point correspondences at known

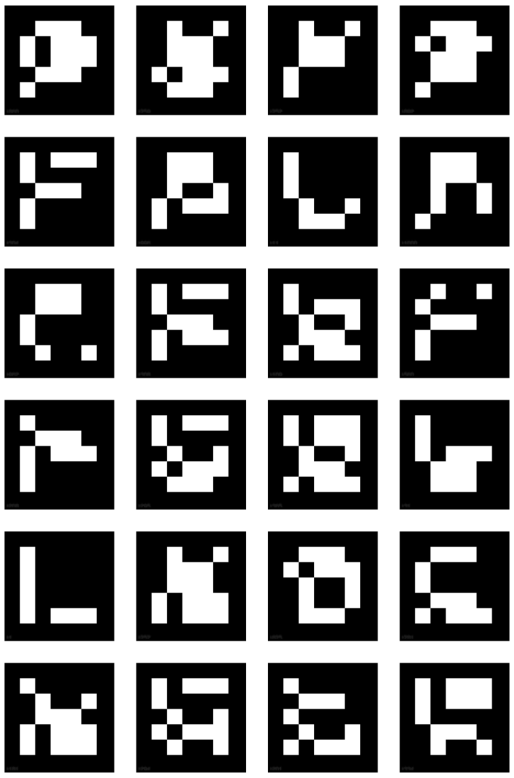
Figure 1. Raw calibration board image

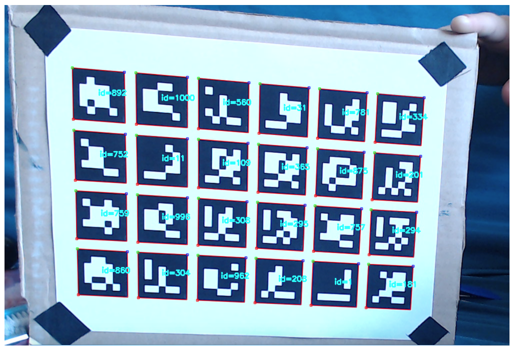
Figure 2. image from webcam during calibration process

board z-positions was used to solve for the camera parameters matrix.

3.3. Marker Detection

Once the camera has been calibrated, we can attempt to extract the markers from the image. We do so first by using the Canny edge detection algorithm, which works as follows: We apply a Gaussian filter to the image to smooth it, and proceed by applying gradient convolution masks in the

x

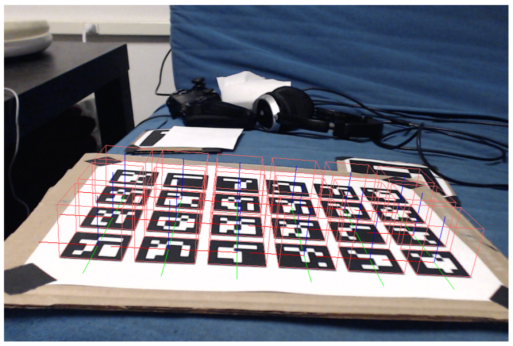

Figure 3. Pose and position of markers on the calibration board

and

$$y$$

directions to get the corresponding gradients of the image. We then combine these to come up with the magnitude and angle matrices for the image using the following:

$$mag = \sqrt{x^2 + y^2}$$

$$\theta = \arctan \frac{y}{x}$$

Next, we use non-maximum suppression to shrink the edges into thin lines. In the final step, we use a high threshold to consider the start of an edge, and a lower threshold to allow for an edge to continue, when identifying whether a set of pixels is part of an edge. For performance reasons, the OpenCV Canny Edge Detector implementation was used.

We then leverage the OpenCV findContours method to come up with contours from the edge information gathered by Canny. We remove all candidates contours with too few contours, too many, and contours that have too few points. Rectangles that are overlapped by other rectangles are also removed. We then finally choose the most external border in the sets of rectangles we found to find the marker edges. This was initially implemented in python using OpenCV cmethods for each step, but for performance reasons we opted for the optimized pipeline in the ArUco library.

Having found the marker positions, we use the OpenCV findHomography method to convert our skewed marker edges into a square image that we can partition into a grid to identify white and black squares. Finally, given the grid of white and black sections, we can identify each marker [4]. As a last step, we use the corners of the markers as 4 points to solve for the pose and position of the marker. This is using the OpenCV solvePnP method, which uses 4 point correspondences, as well as the camera calibration matrix, to solve for the rotation matrix and translation vector associated with those 4 points. An example of this is shown in figure 3 with the calibration board.

3.4. World Localization

Given the pose and position of each of the markers, we now create our virtual world model. Each of the markers has an ID associated with the pattern they contain, and we chose the marker with ID 3 as the origin point for the world. We use the other markers in the world as the positions for which objects are placed in the simulation.

We compute the position of the objects associated with non-origin markers in our virtual world using the pose and position matrices for each marker. If M_o is the matrix that transforms a point from the origin of the screen in OpenGL coordinates to the origin marker in world space, and M_i is the matrix that transforms the origin of the screen in OpenGL to marker i in world space, we can do the following:

$$origin_{world} = M_o * \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$x_{world} = M_o * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} - origin_{world}$$

$$y_{world} = M_o * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} - origin_{world}$$

$$z_{world} = M_o * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - origin_{world}$$

$$P_{iworld} = M_i * \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - origin_{world}$$

$$P_{ix} = x_{world} \cdot P_{iworld}$$

$$P_{iy} = y_{world} \cdot P_{iworld}$$

$$P_{iz} = z_{world} \cdot P_{iworld}$$

We project out all the points we care about into the OpenGL world space, as well as the x, y, and z axes for just the origin marker. We then subtract out the position of the origin marker from all of these, giving us vectors relative to the origin marker to its axes and the other markers. We then take dot products of the desired marker points and the origin axes to get expressions for the positions of these points in the frame of the origin marker. We feed this information to the simulation engine as the positions of non-origin markers so that we can accurately render and simulate the physics of these objects.

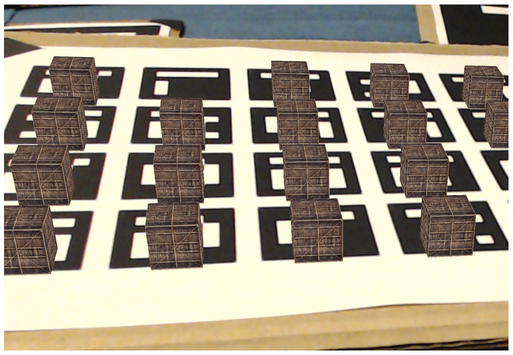

Figure 4. World simulation running, rendering 3D mesh boxes on marker locations based on origin

3.5. Simulation and Graphics

We implemented a simple ”game” engine akin to those used for simple 3D games in order to make use of the vision pipeline information, organize the data structures we were using, and maintain the timestep of the simulation and visualization. On each loop, we fetch a frame from the camera and perform the marker detection and pose estimation. We then compute how long it took the game loop to run from its last execution start, and run the physics simulation for that long, as well as any logic updates. We then extract the position and rotation information from the physics simulation for each of the objects we are simulating, and feed these into custom OpenGL vertex shaders, along with M_o, the transformation from the origin in OpenGL space to the point where the marker exists in world space. For each vertex we want to render, we compute its position as follows:

$$p' = P * M_o * T * p$$

Where p is the location of the point in the model, p' is the position of the point in OpenGL space, T is the transformation matrix extracted from the physics simulation, M_o is the model-view matrix we use to transform points to their correct position in the world, and P is the projection matrix that we calculate from the camera intrinsic parameters we obtained during calibration. See figure 4 for an example of the basic rendering process.

A simple OpenGL fragment shader without lighting modeling was implemented to render textured models as well, and was verified using a wooden box texture. The physics simulation was initially implemented for 2 dimensions assuming a planar world, but was later swapped out for an off-the-shelf engine capable of handling 3D physics with box colliders.

Finally, a separate OpenGL shader is used to render the camera image onto the back of the OpenGL world. A textured quad is placed there, and the camera images are converted into texture buffers which are then sent to the GPU so

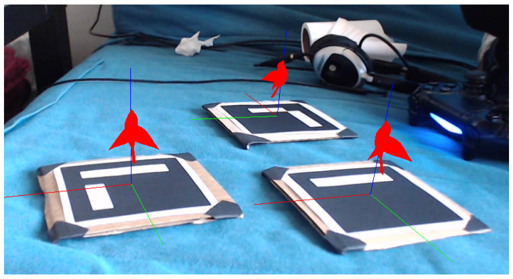

Figure 5. World simulation running, rendering 3D mesh boxes on marker locations based on origin

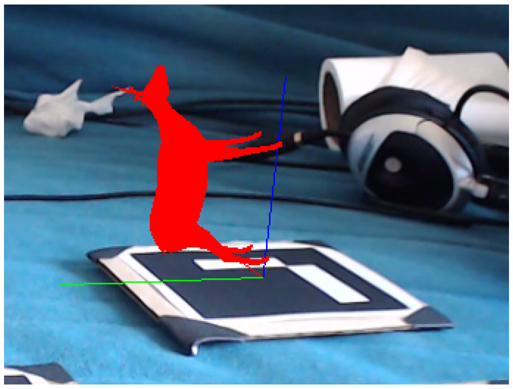

Figure 6. World simulation running, rendering 3D mesh boxes on marker locations based on origin

that OpenGL can make use of them to draw them, to make it appear that the objects being rendered appear on top of the markers that control their locations.

4. Experiments

We ran the simulation program on a variety of different markers, with cameras at different positions and angles relative to the markers, and using different 3D models. We first ran the simulation environment without any physics to test the rendering engine with simple models and no textures, to make sure that vision information was being captured properly and sent forward through the pipeline. The results of this are visible in figures 5 and 6.

Next, we test the full pipeline with collisions enabled on simple 3D boxes, with the camera at low, medium, and high angles to the planes containing the markers. Figures 7, 8, and 9 demonstrates that the results are consistent, and the vision information is propagated forward in a useful manner to effect the simulation. In the video presented in the introduction, we can see that a user can move a marker around and effect the simulation world in real time.

The application was tested using both a Macbook Pro camera, as well as a Logitech C920 web camera. Performance was better with the web camera, in part likely due

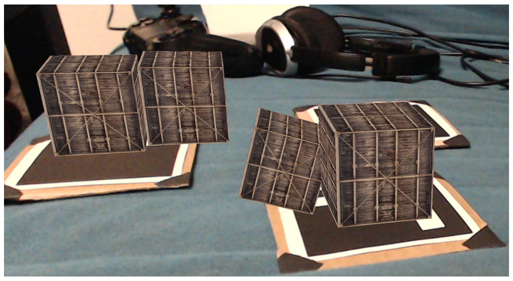

Figure 7. World simulation running, rendering 3D mesh boxes on marker locations based on origin

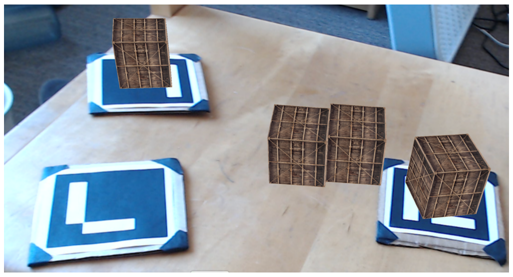

Figure 8. World simulation running, rendering 3D mesh boxes on marker locations based on origin

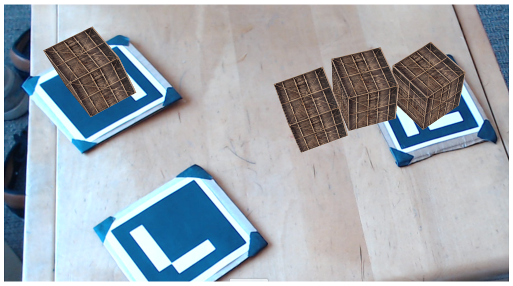

Figure 9. World simulation running, rendering 3D mesh boxes on marker locations based on origin

to its higher resolution, and smaller distortion coefficients when comparing the calibration values obtained for the two cameras.

Both setups were tested in various lighting environments, and unsurprisingly, the marker detection algorithm did not work well on low-light images. Because accurate edges need to be found from image gradients at the start of the process, poor lighting inhibits the application at the very beginning. Different camera angles were tested as well, with much more success - as long as the marker is not occluded, the marker detection phase will find the marker regardless of skew, and often properly detect the marker ID as well.

5. Conclusion

The application presented here showcases a novel way of applying augmented reality techniques to enhance user experience and interactions. It is built of a pipeline that uses both vision techniques, as well as computer graphics and game development methods, to create a more tangible example of the things augmented reality can do. The result is expandable to many other potential applications, games, and interactive demos, since it develops the groundwork and connects the underlying systems in a way that is not exclusive to the presented example.

One of the biggest challenges with this project was fitting the computation into a short enough amount of time to allow for real-time rendering. An augmented reality application that cannot function in real-time is effectively useless, since a user cannot move around and experiment with repositioning the markers. Some features, such as lighting for textures, as well as a homebrew implementation of the marker detection algorithm, were scrapped due to the timestep constraint.

Combining the different pieces of the pipeline was also a source of challenge - the values returned by OpenCV detection were not quite right when sent directly to OpenGL, due to the scaling of the pixel space being different, as well as the positioning and alignment of the coordinate axes. The y and z axes specifically are flipped, and this was a source of a lot of painful debugging and confusion. Designing the data structures which held the information relevant to each physical object in the simulation also required a large amount of code and time, just because the information that controlled their positions and interactions was coming from multiple sources. Finally, while ideally I would have finished writing a simple 3D physics engine myself, there wasn't enough time to build one that was fast enough and robust to use in the application.

5.1. Links to Content

Youtube: https://youtu.be/pHEf27VqGBA
Github: https://github.com/tediris/CS231AProject

References

[1] G. Bradski. *Dr. Dobb's Journal of Software Tools*.

[2] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.

[3] S. Garrido-Jurado, R. M. noz Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.

[4] S. Garrido-Jurado, R. M. noz Salinas, F. Madrid-Cuevas, and R. Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51:481 – 491, 2016.

[5] W. E. Mackay. Augmented reality: Linking real and virtual worlds: A new paradigm for interacting with computers. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '98, pages 13–21, New York, NY, USA, 1998. ACM.