# Executive Summary

The purpose of our 3D scanner for Unity is to provide a useful and novel tool for the Unity game engine. This application will be free to use for UCF students and potentially added to the Unity Asset Store for use by indie game developers. The only similar application that is currently available is Intel® RealSense™, which does not offer the 3D scene segmentation features to help developers with level creation that we are planning.

The scanner will consist of three major modules, the first of which accepts RGB-D images from sensors, like the Microsoft Kinect, and preprocesses the data to prepare it for the interpreter. The data interpreter uses this data as input for a computer vision system that will run a scene understanding algorithm to detect objects and estimate object poses in 3D space. The last module will take the information gathered from the computer vision system and transfer this into a format that can be ported into Unity. Then this module will render the appropriate models in a Unity scene.

# Overview

## Broader Impact

What we are creating is more than just a tool to build prototypes of videogame levels. It breaks down many of the barriers of entry to videogame development, softens the learning curve of game design in general, and makes game design accessible to a wider audience. One of the barriers to video game design is the time required to build a game. To build anything of reasonable complexity, a significant investment of time is required to both design the level and then implement it. Our tool aims to consolidate the design and implement stages into a single step. By doing so designs can be quickly evaluated, modified, and revaluated to arrive at the best course of action in as little time as possible. This significantly lowers the barriers of entry to small game studios and single person teams for creating high quality games. This tool will allow them to develop higher quality games without requiring the resources that large game studios have. Our tool also softens the learning curve for learning how to create videogames. The tool eliminates the need to learn any new skill to design levels. This allows individuals who are interested in learning about game design to complete initial projects faster and more quickly evaluate how they feel about the field of videogame design in general. Finally, our tool makes game design more accessible to those who would otherwise not be able to develop videogames via traditional means. By using blocks to design levels rather than writing code or using a two-dimensional drag and drop interface, people with underdeveloped computer skills can engage in videogame design. This means that young children, elderly, and those lacking finer motor skills would be able to play levels of their own creation.

## Personal Motivations

### Brandon Aulet

### Timothy Flowers

I first became interested in working on this project because of the interest I have had in videogames. Although I have never had a very strong desire to work in the video game industry, I have always enjoyed videogames as someone who plays them. When I began my course work as a Computer Science major, I also began to appreciate them on a technical level. When I saw the pitch for this project I thought it would be a great opportunity to exercise both my passions in videogames as well and technical programming.

Apart from my love for videogames, I’ve also been heavily interested in graphics. I think the field presents unique programming opportunities and paradigms that are not often encountered in other areas of computer science. These opportunities include the usage of highly parallelized graphics cards as well as the design of shaders which model the behavior of light when it reflects off different surfaces. I felt like this would be a good project to further explorer my interest in graphics since there would be opportunity to process three dimensional models as well as write an application that interacts with the Unity game engine.

Finally, I found the initial concept of the tool very exciting. When the implementation is complete, it will allow game designers to eliminate the need to spend so much time prototyping levels. Our tool should give them the ability to test an initial concept and then begin to build on top of it. I get a certain satisfaction at building tools for other people to use because I feel that in a certain way, I’m responsible for what people can create with it. To be able to design something that can help others complete a task much more quickly and efficiently than would otherwise be possible is what software engineers should always strive to do.

### Mark McCulloh

### Christopher Williams

I had previously suggested a Senior Design project similar to this, but utilizing procedural generation techniques. I wanted to assist game developers in level creation by creating procedurally generated assets/levels as a tool for the Unity or Unreal Engine. This project gives the same satisfaction in assisting game developers increase efficiency in level prototyping and will allow me to work with the Unity Game Engine as I intended.

I had not considered that my experience with Computer Vision could help with game development and I am excited to apply my experience in this field and learn much more. I am already familiar with resources for potential previous implementations of Computer Vision systems in 3D scene reproduction. I felt that this opportunity could open a door to future game development positions and combines two fields that I am passionate about.

# Specifications

## Goals

## Requirements

# Research

## Camera Research

### Available Cameras

The UCF Games Research Group had several devices available to us for no charge. These included: Intel® RealSense 3D, Microsoft Hololens, HTC Vive, and Microsoft Kinect. The following is an analysis as to the suitability of each of the devices

#### Intel® RealSense 3D

The Intel® RealSense 3D camera is a small rectangular camera that could easily be mounted in a variety of settings. The camera provides the ability the obtain both color streams and depth streams. Its SDK includes not only the tools to interface with the device itself, but also prebuilt algorithms for 3D scanning and other computer vision applications. The only drawback to the device is that it must be tethered to the computer via USB. This could make it difficult to capture all the necessary angles for the construction of the Unity scene.

#### Microsoft Hololens

The Microsoft Hololens is an augmented reality headset that projects images onto the viewing lens to make it appear as if the images are sharing the same space as the user. Unlike the HTC Vive, Microsoft Hololens does not remove the user from the environment. This allows the user to stay visually engaged in the environment and move about safely. The Hololens also has the added ability of being untethered which allows for easy movement independent of the location of the device running the rest of the application. The primary drawbacks to the use of the Microsoft Hololens are: battery life, cost, and usability. While in use the battery only lasts for approximately two hours. Returning the device to full charge requires approximately five hours. This does not coincide with our desire to create a tool for rapid prototyping. While the untethered design is desirable, it does not justify the sacrifice to be made for battery life. The cost of the device is also prohibitively expensive. It would not be a resource that could be acquired easily by small game studios or individual developers. To maintain the accessibility of our tool a more cost effective device is needed. Finally, we considered the use of a head mounted device impractical for our project. The benefit of allowing the user to capture the spatial data hands free is not significant since the user's hands would have to be cleared from the workspace before capture could begin. This means that the user cannot perform any actions with their hands while capturing the data. This makes the hands free capability of any head mounted headset insignificant for our project.

#### HTC Vive

The HTC Vive is a virtual reality headset. Although it does have spatial scanning capabilities, it completely removes the user from the environment they are working in. This does not make it suitable for this task since it requires visual presence to place the blocks on the scanning surface as well as awareness of the environment to perform the actual scanning of the blocks. The cost of the device also makes it prohibitively expensive and is not congruent with the accessibility that we desired our tool to provide.

#### Microsoft Kinect

The Microsoft Kinect is a rectangular sensor that can provide both depth and color data. Much like the Intel® RealSense 3D camera, its SDK also includes prebuilt computer vision algorithms in addition to the standard camera interface functionality. It also shares the disadvantage of needing to be tethered via USB to the main computing device. The current mode of the Kinect sensor has the additional disadvantage of needing an adapter for use with a laptop. This increases the cost of the device as well as marginally increasing the complexity of the set up for the user.

#### Final Decision

Our main decision was choosing between the Intel® RealSense 3D Camera and the Microsoft Kinect. Both sensors had many of the same advantages and disadvantages. The differentiating factor between the two was the size of the sensor and the cost of the sensors. The Intel® RealSense Camera was marginally cheaper and we felt that its smaller size provided us with more flexibility as to mounting options. The primary benefits we saw the camera providing were the affordability of the device, the included API, and the handheld usability. The device costs approximately $100, which achieves a greater level of accessibility that we wanted to provide with our tool. The handheld usability means that camera can be aimed easily and moved around the workspace as needed. Although the USB tethering of the device could make certain angles difficult, the user of a rotating platform or a mobile computing device could be used to minimize this difficulty. The use of such solutions would allow images to be captured from every angle which is necessary for the computer vision algorithms that we will implement to process the data.

### Camera Module Implementation

There are four choices of implementation for the Camera module of our application. They are C# .NET4, C# Unity, C# UWP, and C++. The C++ implementation provides a native interface for the camera and the other three implementations are wrappers for the C++ implementation. The four different approaches are described and analyzed below.

#### C# .NET4

This implementation allows for the .NET4 Framework to interface with the Intel® RealSense camera. We would create a DLL file that provides access to the data that we wish to retrieve from the camera. This implementation provides the benefit of allowing me to draw on my previous .NET development experience. The implementation provides the benefits of a managed language so no extra time would be spent managing resources. The primary downsides to this approach would be the complexity associated with calling an external DLL from Unity, and the performance loss of using a managed language over a native one. The additional complexity of using an external DLL should be very minor since there are only a few points of interaction between the Unity game engine and the external DLL. The performance loss of using a managed language should not be of a great concern for this application. The application is not performing real time data analysis so the need for the extra performance is not great. The module’s primary responsibility is to gather data from the camera and possibly do some light preprocessing. Neither of these tasks are time critical so the need of a native implementation is not necessary.

#### C# Unity

This implementation allows the camera module to be written directly into the plugin. One of the benefits of implementing the camera module within the Unity plugin, is the reduction in complexity of the project. There will not be the need for an external DLL for the camera module and will simplify the structure of the project. Another benefit to this approach is the automatic memory management that a managed language provides. This reduces the complexity for the programmer, allows for faster development, and is less likely to introduce common errors such as memory leaks into the project.

#### C# UWP

This implementation allows for the Universal Windows Platform to interface with the Intel® RealSense camera. This would involve creating a UWP app that would interface with the camera and then transfer the image data to back to Unity as a saved file. The benefit of creating a UWP application is that the camera module can be run on all variants of Windows 10. This means that the module could run on phones without any code changes if necessary. Since there is no benefit to being able to run the camera application on a phone, the benefits are not useful to our project.

#### C++

This implementation does not use a wrapper and is a pure native implementation. This gives the advantage of a boost in performance but also means that we must handle our own memory management. This module of the project does require a high level of performance. The camera module is not required to do any real-time data processing and the amount of processing that is done in this module is relatively light. The advantages of a native implementation are not as significant here as they would be in other applications.

#### Final Decision

For this module, it makes the most sense to use the C# Unity implementation. Since C# is managed, the code required is simpler and less prone to errors being introduced by the programmer. The lack of real-time processing in the camera module also means that the negative performance impacts associated with managed code are reduced. Finally, since the code is incorporated directly into the Unity plugin, the need to call an external DLL is eliminated and will make the deployment and maintenance of the project simpler.

## Computer Vision Research

### 2.1 – Previous Methods

We have studied many state-of-the-art computer vision methods for 3D scene processing, object detection, object recognition, and model alignment. Our goal with this research is to find a method or methods to adapt for our application that will provide a fast, accurate, and robust method of processing a 3D scene from our camera and exporting usable information to the Unity Game Engine to create a template level layout for the user.

We ensured our search was broad and included as many different methods as possible to allow for the mitigation of any single method failing or not satisfying the needs of the user. All of the following methods will require significant refinement and alteration to meet our needs but will save us time overall because we will not have to develop a 3D computer vision algorithm from scratch.

Most of these methods provide bounding box information as output after processing. If rotational information is not provided this bounding box gives us the ability to infer where objects are in the scene and allows us to convert this information into a 3D box primitive as our input into the Unity Game Engine. This would work for a physical level built with only rectangular blocks, but we would like to find a method robust enough to include other types of blocks such as cylinders, cones, and pyramids. If a 3D model of the object and sufficient rotational information is provided we can fit other block types within the bounding box. With the appropriate rotations applied this provides a successfully and robustly matched object in the 3D scene space. Other methods match pre-existing 3D models to specific data points in the scene provided.

The limitation set by these model-matching methods would be that users must use these specific types of blocks to get accurate results from our software. This will satisfy our project requirements, but will not make a robust system for broader use. A stretch goal would be to implement more robust methods for alignment that do not rely on pre-existing models. For now, we will adapt one of the model-alignment methods for our software. Any of the methods that require 3D models are appropriate for our purposes because we have been provided 3D models for each of the block types present in our target block set.

#### 2.1.1 - Learning 6D Object Pose Estimation using 3D Object Coordinates

#### 2.1.2 - Aligning 3D Models to RGB-D Images of Cluttered Scenes

#### 2.1.3 - Deep Sliding Shapes for Amodal 3D Object Detection in RGB-D Images

#### 2.1.4 - Uncertainty-Driven 6D Pose Estimation of Objects and Scenes from a Single RGB Image

This paper, which debuted at the 2016 Computer Vision and Pattern Recognition (CVPR) Conference, by Brachmann *et al.* is currently our most useful resource for the computer vision interface of our software. The paper is packaged with source code and extensive documentation which allows us to study in-depth what their method is accomplishing and how it functions. This allows us to accurately weigh the benefits and restrictions of this method in comparison to the other methods reviewed.

This algorithm begins by predicting object coordinates and labels with a modified random forest called a joint classification regression forest.

Then Brachmann *et al.* use a stack of forests to generate context information for each pixel in the input image(s).

The object poses are then estimated using Random Sample Consensus (RANSAC). This method is able to perform multi-object detections by obtaining pose estimations for multiple objects and deciding which object the estimations belong to during processing. This is done with the initial predicted values on the input image.

The poses gathered from the use of RANSAC are refined by calculating the distribution of object coordinates in the input image(s). Then the uncertainty levels previously predicted are used to predict camera and object positions when depth data is not available.

### 2.2 – Inputs

There are two basic input formats for the incoming camera data: Point Cloud Data (PCD) or RGB-D image pairs. Point Cloud Data provides millions of data points which provides an implied high accuracy level. The trouble with Point Cloud Data is that minimization or simplification would be required before processing if we wish to achieve fast runtimes.

RGB-D image pairs would contain an RGB image alongside a depth image per frame. This provides a faster runtime more similar to image processing tasks, but it still provides depth information to make sufficiently accurate processing results for our purposes. For these reasons we have chosen to utilize the ability of the Intel® RealSense camera to capture RGB-D image pairs for our application.

### 2.3 – Datasets

### 2.4 – Outputs

## Unity Game Engine Research

### 3.1

# Detailed Design

## Camera Design

### Public Interface

#### StartCapture

#### StopCapture

#### ImageAvailable

### Sub Modules

#### CameraInterface

#### DataPreprocessor

## Computer Vision Design

## Unity Design

# Design Summary

## Camera UML

## Computer Vision UML

## Unity UML

## Overview UML

# Testing Plan

## Camera Testing

## Computer Vision Testing

## Unity Testing

# Budget

## Camera Costs

The Intel® RealSense™ was already available to the UCF Games Research Group. Therefore the use of the camera will not carry a cost to our group. The only potential cost the camera could pose is if we find the Intel® RealSense™ camera to be unusable and we have to use a camera that the UCF Games Research Group does not already have in their possession.

## Unity Costs

# Milestones

# Summary