Reality Capture Inside Autodesk® AutoCAD® Using Kinect® Fusion
Kean Walmsley – Autodesk
Moderator: Stephen Preston - Autodesk

DV1441: Kinect Fusion allows you to "paint" in 3D using a Kinect device as a wand, aggregating a point cloud representing a 3D volume (and allowing you to export a mesh of the capture surface). Kinect Fusion started as a UK-based research project between Microsoft Research Cambridge and Imperial College London. It was introduced into the Kinect for Windows® SDK in the 1.7 release, and has the potential to enable low-cost, high-quality reality capture for anyone with a Kinect device. This class looks at a prototype integration of Kinect Fusion inside Autodesk AutoCAD software, investigating the quality of the results and the potential for further development.

Learning Objectives
At the end of this class, you will be able to:

• Explain the capabilities of the Kinect SDK
• Explain the concepts implemented by Kinect Fusion
• Use C# to integrate Kinect Fusion into AutoCAD
• Capture 3D models inside AutoCAD using Kinect Fusion

About the Speaker
Kean has been with Autodesk since 1995, working for most of that time in a variety of roles—and in a number of different countries—for the Autodesk Developer Network organization.

Kean’s current role is Software Architect for the AutoCAD family of products, and he continues to write regular posts for his popular development-oriented blog, Through the Interface. Kean currently lives and works in Switzerland.

kean.walmsley@autodesk.com
Introducing Kinect

Since Kinect for Xbox 360® was launched on November 4th, 2010, the device has taken the world by storm: it became the quickest selling consumer electronics device ever (according to the Guinness Book of World Records), selling 8 million units in the first 60 days. This record has since been surpassed, but still.

Kinect was originally intended to be a controller for the Xbox 360 gaming system – allowing you to play games without a controller, or, as Microsoft like to say, you are the controller – but it soon became clear that this type of technology has much broader applications.

How Kinect Works

Active scanning technology – such as laser scanners, for instance – typically work on the basis of “time of flight” measurement: they emit pulses and measure the amount of time it takes for the reflected pulse to be returned, much in the same way as radar or sonar works. The technology behind Kinect 1.0 – based on technology licensed by Microsoft from PrimeSense – is different: it does not measure and use the time of flight, rather it projects a specific (yet seemingly random) pattern of infra-red dots and checks any deformation of the pattern, effectively determining the topography of the objects (or people) upon which the pattern has been projected. This is known as “structured light” scanning.

In addition to the camera being used to detect the pattern of infra-red dots, Kinect devices contain an additional camera to detect visible light (i.e, it also takes pictures :-). It’s with these two cameras – and some onboard electronics – that Kinect is able to generate depth and RGB images of the scene being captured. These images get refreshed frequently (30 times per second), and can be combined to generate point clouds of the scene.

In addition to this hardware-based scene capture, higher level capabilities – such as skeleton tracking – are provided in software by some SDKs and middleware components. It’s via these components that applications can interpret gestures, for instance.

It’s worth noting, at this stage, that the next iteration of the Kinect technology – due to be first released with Xbox One – has undergone a significant overhaul. While details are still emerging, it appears Kinect 2.0 will now in fact use “time of flight” rather than structured light, allowing it to work in more varied lighting conditions as well as being more responsive. And with a 1080p HD camera, life is set to get really interesting.
The Race to Hack Kinect

On the day Kinect was released in North America, Adafruit Industries announced a competition to “hack” Kinect. They initially offered a $2,000 bounty for the person reverse-engineering a driver – for any OS, with the primary stipulation being that the implementation be shared via an open source license and posted to GitHub. The bounty increased to $3,000 and interestingly was, it later turns out, contributed – at least in part – by Johnny Lee, an engineer working on the Kinect team, at the time. Microsoft misunderstood the intent of the “hacking” competition – apparently believing there was something more mischievous happening – and spoke out against the efforts. They later clarified their initial reaction, and explained (and this is paraphrased) “we wouldn’t have used a standard USB connector if we didn’t want this to happen”.

On November 6th – two days later – the first “hack” was demonstrated, although the coder, AlexP, somewhat controversially chose not to share his code, but to use it to start a commercial venture (it became the Code Laboratories NUI Platform). He did offer to release the code as open source, should the donation fund he set up receive $10,000 from interested parties.

On November 10th, Héctor Martín uploaded his “hack” to GitHub, creating the libfreenect toolkit (now at the core of the OpenKinect community’s efforts) and effectively winning the bounty. AlexP decided to close the donation fund and, in turn, donated the $457 so far contributed to Héctor, to further reward and encourage his efforts.

On November 14th, Oliver Kreylos, a researcher at UC Davis, posted a couple of videos to YouTube, showing the ability to capture 3D data coming via Kinect and how to measure the device’s accuracy. Oliver had written his own drivers, but had based the work on that performed by Héctor.

The first video went viral. Within a matter of days it had received 1 million views and for that period was the most watched video on the whole of YouTube.

At this stage in the game, we now had a few low-level drivers providing access to the depth and color data generated by Kinect. The availability of these drivers led to the creation of a whole slew of “Kinect Hacks”. KinectHacks.net, launched on November 12th, became the “go to” destination for such hacks, and – it being the Internet – a number of copycat sites eventually spawned around it. This became the place that aspiring “hackers” would post the fruit of their efforts and earn the respect of the broader public.

In early December, PrimeSense – the company that had licensed much of the 3D capture technology to Microsoft – founded the OpenNI (for Open Natural Interaction) community. They released the OpenNI SDK as an open source component to allow the use of such depth camera technology. This SDK could be used, along with a “hacked” SensorKinect module (which was also open source), to access the data coming from Kinect. In addition to this, OpenNI also
released the NITE middleware providing higher-level capabilities such as skeleton tracking and gesture recognition.

While PrimeSense licensed certain technology to Microsoft – and may well have done so at an algorithmic level, also – it’s clear that the core Xbox skeleton tracking is different to that implemented in NITE: if only for the fact you need to strike a calibration pose for NITE-enabled applications to detect you.

It was clear Microsoft had their own software stack related to Kinect, and on June 16th, 2011, they released the first Beta of their own SDK for non-commercial use. The availability of this SDK greatly simplified the configuration needed to harness Kinect in custom applications on Windows, at least, and it is this implementation that is being used in the code in this session.

The availability of technologies allowing people to harness the ground-breaking capabilities of Kinect in their own applications has led to an explosion of creativity. The variety and ingenuity of the “hacks” made available are breath-taking, and this is really just the beginning. It’s clear there are a number of very compelling – as well as downright fun – use cases for this technology, and many of them relate to the world of 3D design.

The Kinect SDK

The official Kinect for Windows SDK has gone through a number of revisions in the 2+ years since its release. Here’s a brief summary of the major enhancements that have been introduced with each update:

1.8  Background removal, Webserver support, color in Kinect Fusion
1.7  Kinect Interactions, Kinect Fusion
1.6  Windows 8 (desktop) support, accelerometer APIs, infrared emitter control and stream API, VM support
1.5.1 & 1.5.2  Minor Developer Toolkit updates
1.5  Kinect Studio, near-mode skeleton tracking (inc. seated mode), face tracking
1.0  Multi-sensor support, Xbox-equivalent skeleton tracking, near-mode
Kinect Fusion

Kinect Fusion started as a UK-based research project between Microsoft Research Cambridge, Imperial College London, Newcastle University, Lancaster University and the University of Toronto.

The concept behind Kinect Fusion is to take subsequent depth and image frames from the Kinect device and aggregate the data into a volume, allowing a mesh to be extracted, once complete. Without this aggregation step, the depth data provided by Kinect is only ever going to provide a limited view onto an object: the point cloud will be directional and suffer from occlusion.

Since version 1.7 of the Kinect SDK, Microsoft have included the Kinect Fusion component in its excellent Developer Toolkit, something of considerable interest to developers of 3D modeling software with an interest in reality capture.

How Kinect Fusion Works

Firstly, it’s important to note that Kinect Fusion currently works on a fixed volume of space. The maximum size of this volume will depend on the memory available on your graphics card – as in order to achieve the required performance, Kinect Fusion uses the GPU to parallelize many of the operations it needs to perform – as well as whether you’re choosing to capture color (which will approximately double the amount of information needed to be stored for each voxel or volumetric pixel).

Once you have defined the dimensions your volume, the application loop can start that will map out its contents.

The Kinect Fusion sub-system takes depth and color frames from the Kinect device and attempts to use them to flesh out the contents of the volume being mapped. An Iterative Closest Point function is used to attempt to match the overlapping point clouds and work out what’s different: the point clouds need to be roughly aligned for this to work, of course. This means it’s important for the frame-rate to be maintained at a high level for the gap between frames to be minimized and the tracking to continue. More on this later.

Once this complex alignment process has completed and the delta between the frames is established, this data is used to populate the volume with voxel information indicating the boundary of the 3D contents. This repeats multiple times per second (up to the 30fps frame-rate
provided by the Kinect sensor). The volume can then be used to either generate a raw point cloud or a mesh, whether interactively or once the capture process has completed.

Typical applications want to ask the Kinect Fusion runtime to generate a rendered view onto this volume – the ray-casting operation to generate the rendered bitmap is also performed highly efficiently on the GPU, and it’s by far the quickest way to provide visual feedback to the user.

With a 3D application however, this is problematic: you can either attempt to match the view into the volume with that of the 3D scene that’s being displayed – and overlap the image onto it – or we need to get 3D information in real-time from Kinect Fusion. This means we either need to extract the raw point cloud or generate one using the vertices of the generated mesh, both of which have some performance and data marshaling overhead (especially as we want to do this as many times per second as we can).

The 3D application then needs to have APIs that allow this data to be displayed to the user dynamically, providing a sense of what has been captured and where the gaps are. This dynamic capability is not inherent in all software, which is one of the reasons I’ve chosen AutoCAD for this rather than a tool such as ReCap Studio.
Integrating Kinect Fusion into AutoCAD

To integrate Kinect Fusion inside an AutoCAD application, the first step is to include the appropriate assembly references into your .NET project.

You can see from this image that in addition to the standard AutoCAD assembly references – AcMgd.dll, AcDbMgd.dll and AcCoreMgd.dll – we also have assembly references from the Kinect SDK.

As a baseline you’re going to need Microsoft.Kinect, Microsoft.Kinect.Toolkit and Microsoft.Kinect.Toolkit.Fusion – the FaceTracking assembly is for a different sample in the collection.

You will also need to copy a runtime DLL across into the AutoCAD program files folder: for the latest version of Kinect Fusion to work on my 64-bit system, I’ve copied KinectFusion180_64.dll from the Developer Toolkit into AutoCAD’s main program files folder.

We’re going to use an AutoCAD jig to display the point cloud we get from Kinect Fusion. The jig’s polypoint capability – which dynamically displays a set of points on the screen – doesn’t easily support color (you could filter points of approximately the same color into different sets and use different calls to display them, but this would be complicated and time-consuming), so in order to maintain performance we live with the limitation of dealing with monochrome point display during the capture process.
You can see from the above image that collecting just a subset of points has allowed us to keep the frame-rate high. This is very important as it gives Kinect Fusion the best possible chance of integrating the new frames into the volume (without losing tracking). It's still possible to move the sensor too quickly, of course, and lose tracking between frames, but the likelihood is greatly decreased with a high frame-rate.

Points can be generated in two ways: by querying the raw points from the volume or by generating a mesh and using its vertices. Both approaches have been implemented in the sample project for side-by-side comparison. I expected that using the mesh would be slower but it turns out not to be the case (at least not significantly). Both approaches support the use of a voxel step, which is very important to us.

Once the capture has been finalized we need to bring in the full point cloud. As these can be quite big, the user is given the option to use a voxel step for the final import, too.

The points are actually brought into AutoCAD in a very low-tech way: we write the points – with RGB data – to a text file and then use the POINTCLOUDINDEX command to generate a PCG file and POINTCLOUDATTACH to place it in our drawing. We might have used the C++ API for generating point clouds, but I made a conscious choice to keep the code accessible to a broader audience and only make use of public .NET APIs.
For the complete source code showing how to integrate Kinect Fusion into AutoCAD with color output, see this blog post:

http://through-the-interface.typepad.com/through_the_interface/2013/10/kinect-fusion-updated-to-include-colour.html

If you have installed the Kinect SDK and the Kinect Developer Toolkit, then you should be able to get this project compiled ready to be loaded into AutoCAD.

Capturing 3D Models Inside AutoCAD Using Kinect Fusion

To make use of the compiled project mentioned in the previous section, NETLOAD it into AutoCAD. At this point you will have a number of new commands available, the most important of which is KINFUSCOL.

It’s recommended to open the kinfus.dwg file from the samples folder into AutoCAD before launching this command as it will give an appropriate view on the data streaming in from the Kinect Fusion runtime.

Setting up the Capture

The KINFUSCOL command requires a number of pieces of information from the user.

It needs the dimensions of the volume to be captured as well as the resolution (in voxels per meter) and the “voxel step” (the sampling rate to use when displaying the volume dynamically). A higher voxel step means fewer points to display during the jig, which means a higher frame-rate and therefore a lower risk of tracking failure.

In general it makes sense to keep the dimensions of the volume you’re capturing as small as you can, as well as choosing a moderate resolution. When you hit the limits of your GPU memory you may well experience a driver crash (and Kinect Fusion sometimes lets you allocate just enough that a crash will happen during capture).

Also play with the voxel step to get an acceptable balance between the display resolution of the point cloud and the frame-rate.

The options entered here will be remembered for the next time you run the command during the current editing session, allowing you to tune them over time.
The Capture Process

It’s important to keep the Kinect sensor fairly steady during the capture process, moving it in small increments. It helps to have a tripod – you can get special tripods for the Kinect sensor, for example – as this will at least allow you to keep it steady for the beginning. The volume is centered in the X and Z directions (relative to world coordinates – Kinect coordinates typically measure Z as depth from the camera) but at 0 on the Y axis. So if you have the sensor on the ground and want to capture a model that’s a meter high, you will need a volume of 2 meters in height.

To help you capture a model it helps to be able to move the sensor far enough. I’d recommend an electrical extension cable for the Kinect sensor (independent of your PC power) as well as a USB extension. This will give you that extra bit of range which can be very valuable.

Here are a couple of captures created using Kinect Fusion, to give an idea of the possible results: