Visual Gesture Builder: A Data-Driven Solution to Gesture Detection

Published for the Xbox One XDK on September 18, 2013
Adapted for Kinect For Windows v2 SDK on July 20, 2014

Abstract

Many successful and innovative applications use gestures as input. These programs span a wide variety of genres, platforms and input technologies, from the touch screen of a smart phone to the full-motion, natural input of devices like the Kinect Sensor. This white paper explains how Visual Gesture Builder, a data-driven machine-learning solution for gesture detection, can be used efficiently to detect even the most challenging gestures with very high accuracy. This technology can make developers more productive and raise the quality of Kinect applications in terms of better gesture detection and reduced latency.

Contents

Introduction

For Kinect applications, it is essential to successfully and robustly communicate a person's intent in a natural way, for this is the very heart of Kinect—"you are the controller." This brings us to the importance of gesture detection.

Gestures

A *gesture* is an action or a motion that is intended to communicate feelings or intentions. For example, when your dog wags its tail at you, this is a gesture with which your dog communicates that he is happy to see you.

Gesture detection

Gesture detection is the ability of a computer to understand human gestures as input. It has been around since 1963, when the first pen-based input device was designed. Gesture detection is still used in many technologies today, such as touch screens, computer mice, handwriting recognition, and Kinect.

Machine learning

When we say *machine learning*, we refer to the ability of a computer to automatically learn to recognize complex patterns in data. A computer usually does this by learning from empirical data examples, and the result is that it can classify data that it has not yet observed. There are many different approaches to machine learning, such as weighted networks and neural networks, decision trees, support vector machines (SVM), clustering, Bayesian networks, and boosting. Examples of where machine learning is used are internet search engines, face detection on digital cameras, speech recognition software, stock market analysis, and artificial intelligence (AI) in games.

Traditional gesture detection with Kinect

Gesture detection using data from the Kinect Sensor is not a trivial problem to solve. As an example, look at the following code that could be used to implement the detection of a punch gesture.

```
if ( vHandPos.z – vShoulderPos.z > fThreshold1 &&
     fVelocityOfHand > fThreshold2 ||
     fVelocityOfElbow > fThreshold3 &&
     DotProduct( vUpperArm, vLowerArm) > fThreshold4 )
{
    bDetect = TRUE;
}
```

This is a very simple detection that can easily work in an optimal environment—but even this simple piece of code has four thresholds that need to be manually found, fine-tuned and maintained. For detections to work reliably for a wide variety of different people in a wide variety of different environments, the code becomes much more complex very quickly as more code is added to handle additional complicating factors. Some of these factors are that different skeleton data is produced when wearing different clothes, or when the Kinect Sensor is at different heights and angles, and some joints might be occluded due to objects in the room (for example, by a coffee table). This is evident in the large number of lines of code that must be written to detect gestures in real world examples.

Table 1. The number of lines of code used for gesture detection in real-world examples.

Detector	Lines of source code
Wave	550
360 degree turn	500
Jump and duck	2000
Crumpling skeleton	2500
Kinect Sports: Boxing	950

Some of the challenges with the traditional approach of implementing gesture detection by examining data from the Kinect Sensor are:

- It's a time-consuming engineering task, requiring many lines of code.
- Kinect data is complex—for example, twenty-five 3D joint positions with low precision.
- Determining the best detection thresholds can be difficult.
- Latency is always added, since previous frames of data need to be examined.

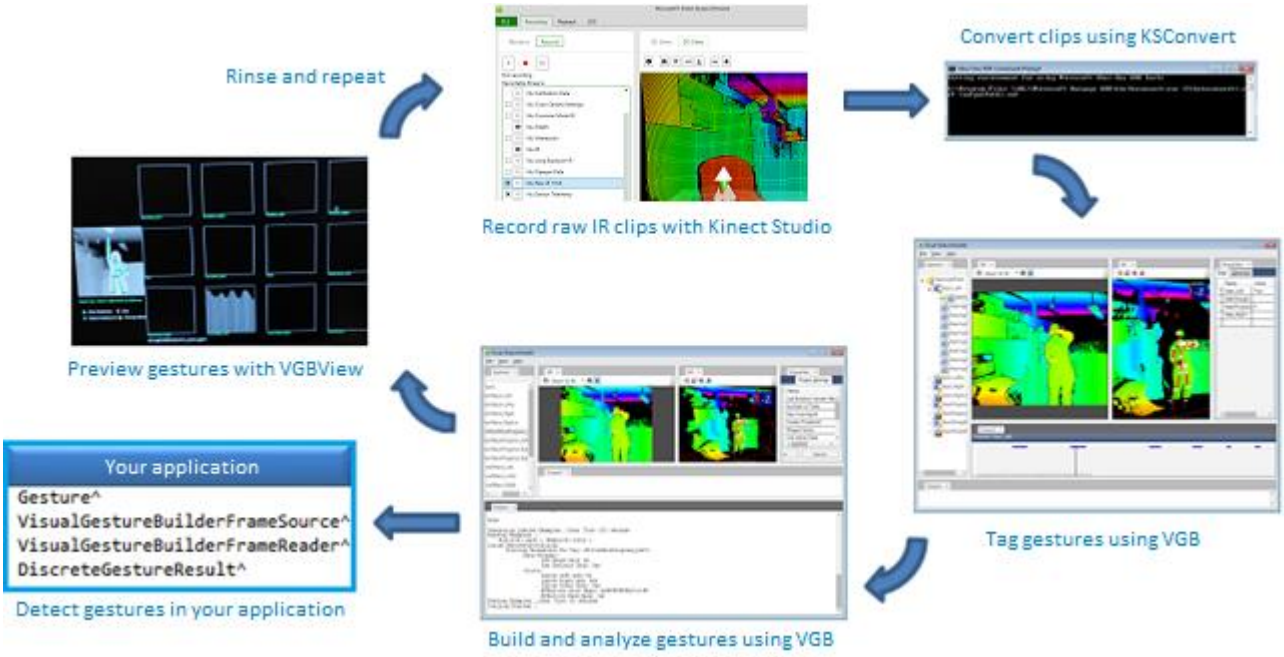
Visual Gesture Builder

Visual Gesture Builder (VGB) is a tool that provides a data-driven solution to gesture detection through machine learning. This essentially means that gesture detection is turned into a task of content creation, rather than code writing, and a task that non-engineers can perform (for example designers, animators, and technical artists). A small gesture database is built using VGB, and using the database has very low run-time costs in terms of memory overhead and CPU processing.

The process of creating a gesture detector using VGB is simple. Firstly, because it is data-driven, you record people while they perform the gestures that you are interested in detecting. Raw recordings can be created by using Kinect Studio, and then they can be converted to processed clips using KSConvert. (For more information about Kinect Studio and KSConvert, see the Kinect SDK documentation). Next, you give meaning to the data by using VBG to tag or label all of the frames in the recordings that define a gesture.

Once tagging is complete, you can build the gesture detector. VGB uses machine learning to build a database that can be used at run time in your title. A live preview tool, VgbView, enables you to quickly iterate over gestures for fast prototyping. For more information about using VGB during run time, see the API reference for the Microsoft.Kinect namespace in the Kinect SDK documentation.

Figure 1. The data-driven process of creating a gesture detector using VGB.

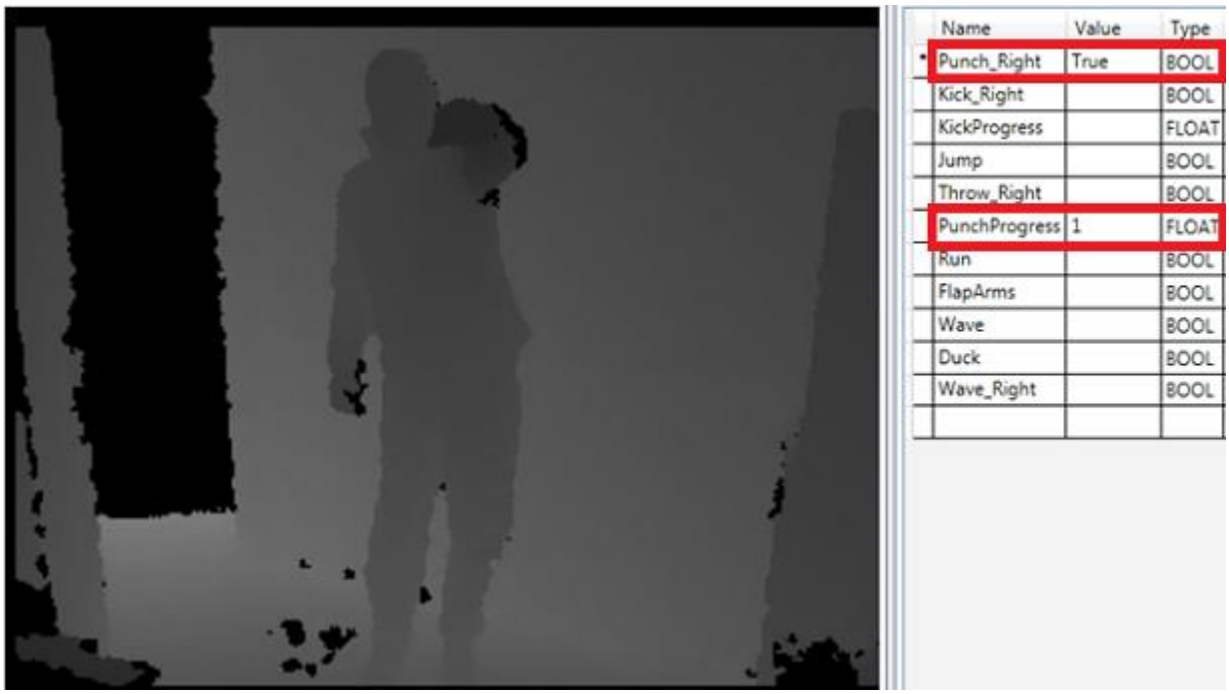


Machine learning technologies

Various machine learning technologies are available in VGB. They are grouped into two categories: discrete indicators and continuous indicators. A *discrete indicator* (for example, AdaBoost Trigger) is a binary detector that determines if a person is performing a gesture and the confidence of the system in that gesture. A *continuous indicator* (for example, RFRProgress) shows the progress of the person while he or she performs a gesture, showing, for example, 50% done with the gesture.

While the majority of your application's gestures will likely be discrete, continuous gestures can be helpful for combining multiple gestures to form a complex whole, such as mapping three discrete gestures—for example, *BackSwing*, *ForwardSwing*, and *FollowThrough*—to one continuous gesture—*GolfSwing*.

Figure 2. "Punch_Right" is a discrete gesture indicating that the person is in the process of performing the gesture. PunchProgress is a continuous indicator, in this frame indicating that this is the last frame of the gesture (that is, 100% done with the gesture).



Unless disabled, continuous gestures will always provide feedback to the program. This signal might be confusing if you don't have at least one discrete gesture to help interpret it.

Discrete and continuous indicators can be combined in creative ways. One example is to use a discrete gesture to determine context. For example a playable character in a game has the capabilities to walk, run, and jump. A simple approach to using gesture detection to control the character is to use discrete gestures to determine when a person is walking, running, or jumping. The playable character can therefore be controlled using gesture input, but only in a binary way—that is, the player is either walking or not.

Combining this with a continuous indicator can add a whole new level of a person connecting with the playable character, making the person feel that he *is* the playable character instead of feeling that he is controlling the playable character. Since the discrete gestures can be used to determine context, only the continuous gesture for the currently active context needs to be run. The results of the continuous indicator can be used to blend character animations appropriately so that when a person is walking slowly, the playable character is walking slowly, which provides a more natural feel to the game.

Collecting data

The recommended process of collecting data is to use Kinect Studio to record Raw IR 11 Bit data and then to convert the files by using KSConvert to the processed event file (XEF) format (that is, processed IR data, depth data, and skeletal data). You should record members of your target audience—individually—while they interact with a prototype of your application in different spaces, under various tilt angles of the Kinect Sensor, and while wearing a variety of clothes.

Data to record

We recommended recording raw files via Kinect Studio to take advantage of any future changes made to the depth/skeleton tracking system. If raw recordings are not an option (due to storage or length limitations), then you must record the following processed NUI streams: Depth, IR, BodyFrame, Opaque Data and Calibration Data. Processed streams can be recorded from Kinect Studio, or by using the **IKStudioRecording** interface.

Whenever improvements are made to skeletal tracking in the SDK, the same depth data will produce different skeletal data. When you record the raw format (XRF), you can convert the file by using KSConvert to generate the latest streams (IR, depth, skeleton, opaque, and so on). The newly converted files can then be used in your training set to generate a gesture database that will work with the latest skeleton/depth tracking system.

In many cases, simply looking at depth or skeletal data won't reveal the intentions of the users; however, by also looking at IR data, users' intentions can be determined. Since this is a data-driven system, it is essential to tag frames correctly. So, make it easier for the people doing the tagging to interpret the recordings by including IR data.

Target audience

Although you need to include a diverse variety of people for generalization and criteria selection in training, it is important to define the target audience of your application. It could be teenagers, small children, adult males, the whole family, and so on. Make sure that this group of people is well represented when doing data collection—don't develop an application targeted at small children and mainly use gesture recordings of adults. Try to include people of various body compositions within your training set, too, because they may perform gestures differently due to their height or weight.

Record people individually

When recording people, record them individually, since people are easily influenced in the way they perform a gesture when they see how others perform the same gesture or when many other people are watching as they perform a gesture. It is best to have recordings of the widest possible range of ways that people perform relevant gestures in their own unique ways, since wide variety is what your application will encounter in the real world.

What gestures to record

Ideally, you record people while they are interacting with a prototype of your application, since people perform a gesture differently when asked to perform a *gesture* versus interacting with an actual program. For example, telling a person to "show me a punch gesture" has a different result than asking "please punch the boxing bag"—people perform differently in these contexts, though the fundamental gesture may be the same.

Also, define the gestures that you want to detect from the user. For example, if you are creating a boxing game, you probably want to detect punches and blocks. These are defined as *positive* examples because they are the gestures that you are aiming to detect. However, and most importantly, you also have to think

of *negative* examples. Determine which motions cause problems for detection, such as false positives, and include them as negative examples for training. For example, a throw gesture might look very similar to a punch, since the hand and elbow move forward in both cases, but the two movements are actually slightly different.

In general, including negative examples in training data reduces false positives, and this leads to a better user experience. Therefore, we recommend that you record many more negative examples for your training data than positive examples. It is usually a good practice to treat all remaining gestures for your application as negative examples during training (for example, a *kick* gesture can be treated as a negative example when training a *punch* gesture).

If you do not have a prototype to use, then try recording a mixture of gesture scenarios. Have the user repeat gestures often and mix their ordering, because the skeleton might behave differently when the person is performing a gesture immediately after another gesture, in contrast to performing the gesture from standing still. For example, if you create gestures for a kick-boxing game, you might make several recordings of the player punching left and right, a set of clips where the player kicks left and right, and a set which includes alternating kicks and punches.

Use a variety of play spaces and sensor tilt angles

Although the algorithms in VGB mostly operate on tilt-corrected skeletal data that is local to the user's space, it is important to record people in different environments, and with different heights and tilt angles of the Kinect Sensor. The main reason for this variety is that skeletal tracking produces different skeletal data at different tilt angles and sensor heights, because body joints are observed differently by the Kinect Sensor; for example, a joint might be occluded from one angle but not from another. Furniture can also introduce occlusions that can change the result of skeletal tracking. Therefore, we recommend that for each person recorded, the Kinect Sensor be slightly moved from its previous location.

Try to re-create the user's environment, and position the sensor in a logical location (centered above or below the screen). Record the user at various distances from the sensor. If the gesture can be performed in a sitting position, be sure to include a variety of seating options and postures. If your application supports multiple users, then there is a good chance that the user will not be centered while using your application, so be sure to record people positioned to the left and right of the sensor as well.

Variety of clothing

Make sure to record people wearing a variety of clothing—for example, skirts, dresses, tank tops, shorts, pants, and wool sweaters. Wearing different types of clothing can produce different skeletal data, even for the same person. Also, be sure to record people wearing the clothing that your customers wear, rather than only what you are accustomed to seeing and wearing. For example, if you are a developer in Florida, chances are that you see short-sleeved shirts and sandles all year round, clothes that people in Belgium would only consider wearing during summer. However, your application might be used by both Floridians and Belgians alike, so your gestures need to work regardless of clothing type.

Testing data versus training data

It is important to understand that the accuracy of the machine-learning algorithm cannot be measured by testing it on the same data that it has been trained with. In other words, do not use *training* data for *testing* purposes. For this reason, whenever a gesture project is created in VGB, two projects are created: one for building/training and one for analyzing/testing the gesture. An example of how to apportion data into different sets is to use 66% of data for training and 33% for testing.

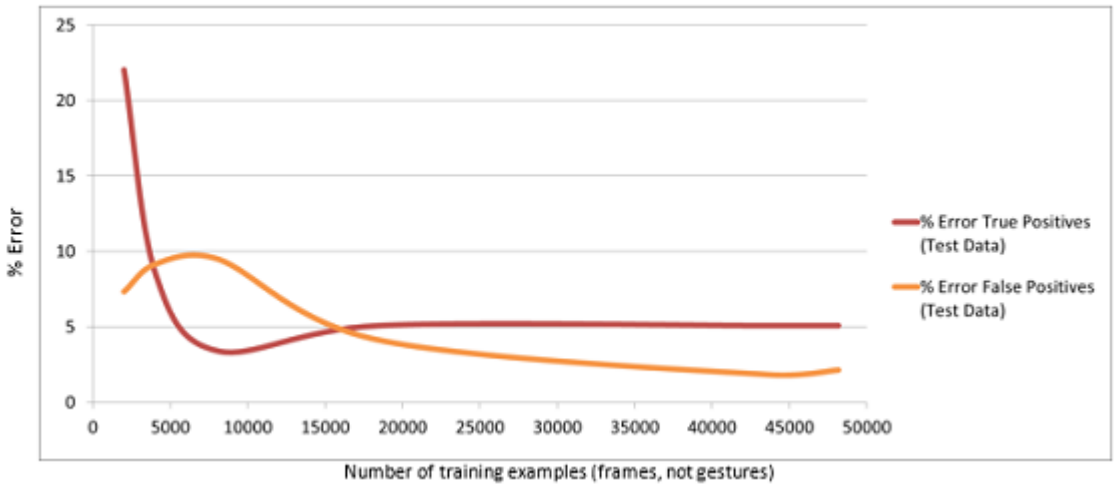
How much data is enough?

When creating a gesture prototype, a small set of training clips (10-20) might prove sufficient, but such a gesture would quickly break down if released to the real world. How much data is required is not a question that has the same answer for every gesture. For example, one gesture might prove reliable with 100 training clips, but a second gesture could require a thousand. So, rather than asking how much training data is needed, the question should be: "How can I establish that I have enough training data for a gesture?"

To answer this question requires counting false positives and false negatives. An example of a *false positive* is when a user performs gesture B, but gesture A is detected. An example of a *false negative* is when a user performs gesture A, but detection fails to identify that gesture A has been performed. The number of these two errors are two different values that need to be interpreted separately—the one is not the inverse of the other. Building and analyzing gestures, later in this paper, explains in detail how to build a gesture, and then how to test or analyze it. The results of this analysis provide the count of errors from false positives and false negatives.

To understand whether you have enough data, you need to track how the values for false positives and false negatives change as you add new training data. In the graph in Figure 3, you can see that with very few training examples, the error rates are high, but as you add more training examples, the error rates go down. The interesting part is that, at some point, the error rates simply stay the same, regardless of how many more training examples are added. When both values for false positives and false negatives start to flatten out at a low error rate, it is safe to assume that you have enough data in the training set. Other measurements can also be used—for example, precision, accuracy, and recall.

Figure 3. Error rate in false positives and false negatives, graphed against the number of training examples, is a good indication of when you have enough training data.



Tagging data

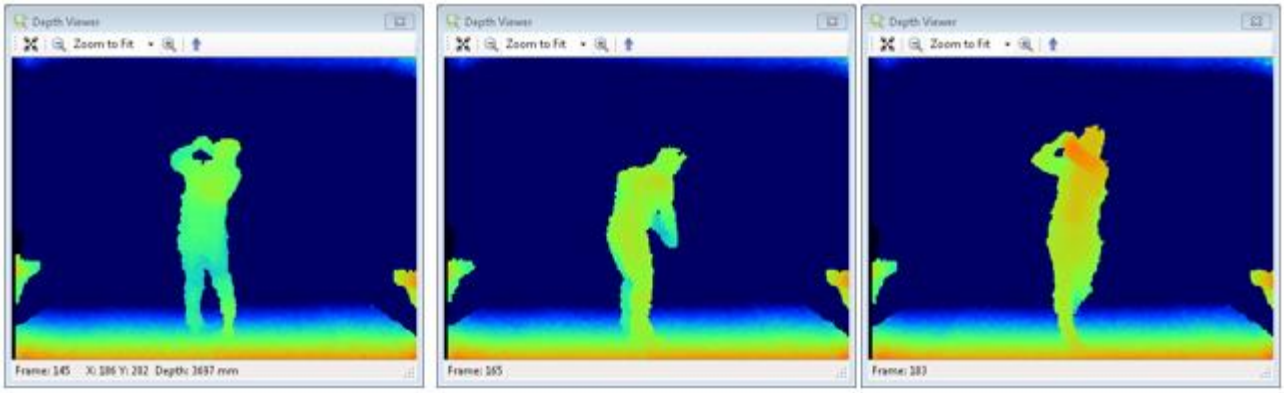
Tagging recorded data is the most time-consuming part of creating a gesture detector with VGB, but it is the most important step. Tagging plays a huge role in the results, since VGB uses a data-driven approach—in other words, *garbage in, garbage out*. The following are a number of best practices to following when tagging data:

- Keep gestures simple
- Avoid over-labeling
- Separate gestures from poses
- Reduce latency
- Be consistent
- Specify Left or Right gestures
- Verify tagging
- Use keyboard shortcuts

Keep gestures simple

Break complex gestures into smaller sub-gestures and build multiple detectors. For example, a golf swing can be broken up into three gestures: a back swing, a forward swing, and a follow through. Detection of cyclic gestures—like walking, rowing, and waving—are almost always more reliable when detection is divided into two simple gestures. For example, to detect a wave, have one detector for when the hand is moving leftward and another for when the hand is moving rightward.

Figure 4. Shows how a golf swing can be broken up into three simpler gestures. The first image shows a back swing; the middle image shows the forward swing where, the golf club makes contact with the ball; and the last image shows the person's follow through.



Avoid over-labeling

Tag all of the frames that make up a gesture, but only tag the core portion of the gesture. Try to find the canonical motions that uniquely represent an action. For example, when tagging a jump gesture, it's best to not include the frames where the user is getting ready to jump (for example, moving downward), but rather, to tag only the frames where the user is starting to move upwards until he reaches the apex of the jump, where his body is no longer moving upward. In general, avoid tagging preparatory or recovery motions.

Separate gestures from poses

Don't mix and match frames from static poses with dynamic gestures. For example if you want to detect a jump gesture, tag the frames of the person in which he or she is moving upward; don't include frames where the person is getting ready to jump or where the person might be in a static pose. Whereas, if your gesture is a static pose, like *glide*, only tag the frames where both of the person's arms are fully extended, not those where the person is moving his or her arms up or down to get into position.

Reduce latency

Latency can be significantly reduced by tagging gestures earlier in a sequence of frames, rather than only tagging when the gesture finishes. Determine the point in the sequence when the person is committed to performing the gesture, and use that frame as the starting point for tagging the gesture. For example, the starting point for a punch gesture could be when the person is leaning forward to punch, but his hand is still close to his shoulder.

Figure 5. An example of how early a punch gesture can be tagged to reduce latency. All these frames include joint information of the person intending to do the punch, therefore all these frames are tagged as a punch.

Be consistent

It is important to tag different examples of the same gesture in a consistent manner; that is, the starting points and the end points should be tagged in approximately the same places throughout the recordings. In some cases, it might be hard to understand the intention of a user by looking only at depth or skeletal data, and in such cases, examining the IR data might make tagging much easier.

Specify Left or Right gestures

When creating a project, an option called **Body Side** can be set to the value **Left** or **Right**. If the gesture to be detected can be performed on either side of the body or in either direction, then we recommended setting a value for **Body Side**; doing so means that your data and tags can be mirrored, doubling the size of your data set for training.

For example, most people are right-handed which means that when recording and tagging a punch gesture, you will have less training data for left-handed people than right-handed people. If your project has **Body Side** set to **Right**, then VGB can use all of the training data from right-handed punches as training data for left-handed punches by mirroring the data.

Figure 6. Shows the "Body Side" option, which specifies on which side of the body, or in which direction, a gesture is performed.

Verify tagging

It is crucial that you verify that data is tagged correctly. Any gesture that is untagged will be used as a negative example during training. This confuses machine learning, because the machine is shown examples of what the gesture looks like (tagged, positive examples), and then it is shown examples of what the gesture does *not* look like (untagged, negative examples). It is also possible that a gesture could be marked with the wrong gesture tag. So, we recommended that a quality assurance (QA) team approve tagged data before it is added to a training set.

VGB can be used to automatically find errors in tagging. The following procedure can find errors where gestures were not tagged or where gestures were tagged with a wrong label.

1. Add all tagged recordings to the build project
2. Add all tagged recordings to the analysis project
3. Build the gesture
4. Analyze the gesture
5. Visually inspect the analysis for
    a. Frames that were detected but not tagged
    b. Frames that were not detected but tagged

Since training and testing is done using the same data (for this type of verification only), we expect the detector to be near 100% correct during analysis. Having any frames where the tags do not match the results of the detector is a strong indication of incorrect tagging. For example, analysis detects a Punch_Left gesture between frames 100-110, but these frames are tagged as a Punch_Right or not tagged at all. Visually inspecting the recording will reveal if the detector or the tag was wrong.

Figure 7. An example of using VGB for verifying correctness of tagging. The analysis indicates that the detector detected two gestures (indicated by the two purple peaks), but three gestures were tagged (indicated by the horizontal blue lines). Visual inspection of what the person actually did indicates that the detector is actually correct, and the tag in the middle is incorrect.

Use keyboard shortcuts

Since tagging is a time-consuming process, make sure that taggers know of the keyboard shortcuts. Using these could significantly reduce tagging time. For a list of the shortcuts, see "Visual Gesture Builder Timeline" in the Kinect SDK documentation. They are also located below the timeline control for quick reference within VGB:

Building and analyzing gestures

After tagging data, you can build either a solution or just one gesture project. During the building process, the appropriate machine-learning algorithm processes the tagged data. This creates a gesture database which can be loaded into your application at run time.

Some interesting information can be seen in the log files. For example, from the log file in Figure 8, you can see that the angle velocity between the spine, shoulder center, and left shoulder is actually a good indicator for a right punch. If you only looked at the top classifier, you can interpret this as:

```
if ( AngleVelocity( Spine, ShoulderCenter, ShoulderLeft ) > 0.5f )
{
    bRightPunchDetected = TRUE;
}
```

Therefore, in the case of the AdaBoostTrigger, you can actually do knowledge extraction from the log files to implement your own gesture detection.

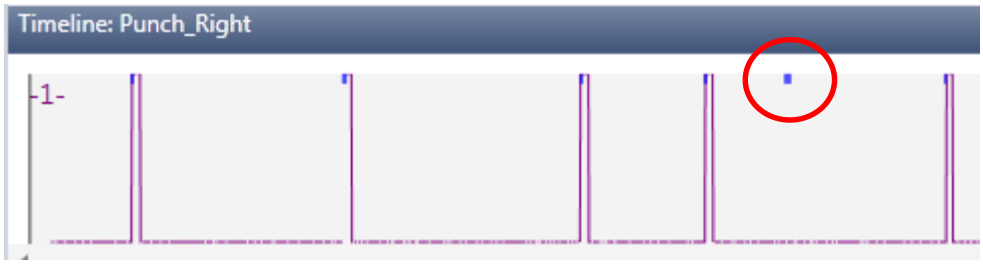
Figure 8. A portion of the log file from an AdaBoostTrigger project that indicates the top 10 weak classifiers defining a particular gesture.

```
Build: Top 10 contributing weak classifiers:
Build:       AngleVelocity( Spine, ShoulderCenter, ShoulderLeft ) rejecting inferred joints, fValue >= 0.500000, alpha = 1.613963
Build:       Angles( Spine, ShoulderCenter, ElbowRight ) using inferred joints, fValue >= 88.000000, alpha = 0.993439
Build:       VelocityY( ElbowLeft ) rejecting inferred joints, fValue >= 1.199998, alpha = 0.784924
Build:       Angles( WristLeft, ShoulderCenter, WristRight ) using inferred joints, fValue >= 84.000000, alpha = 0.651654
Build:       AngleVelocity( Spine, ShoulderCenter, ElbowRight ) rejecting inferred joints, fValue >= 2.750000, alpha = 0.586568
Build:       Angles( Spine, ShoulderCenter, Head ) using inferred joints, fValue < 158.000000, alpha = 0.572036
Build:       VelocityY( ElbowLeft ) rejecting inferred joints, fValue >= -0.000003, alpha = 0.571161
Build:       Angles( Spine, ShoulderCenter, FootRight ) using inferred joints, fValue >= 14.000000, alpha = 0.495747
Build:       Angles( HandLeft, ShoulderCenter, HandRight ) using inferred joints, fValue >= 80.000000, alpha = 0.455011
Build:       AngleVelocity( ShoulderCenter, ShoulderRight, AnkleRight ) using inferred joints, fValue < -0.250000, alpha = 0.445416
```

To test how well the detector fares, you can test it at run time by using **Live Preview** in VGB (available on the **File** menu), which runs the detector in VgbView and is very useful for fast prototyping.

Another way of testing is to use the analysis project. This will run the detector within VGB. The biggest advantage of using the analysis project is that you can compare the results across multiple builds; for example, you can determine if adding more training data really improved the detector or not.

Figure 9. An example of the analysis of a gesture where five tagged gestures (blue lines) are correctly detected (purple peaks) and one is not.



For more information about the analysis project in VGB, see “Visual Gesture Builder Analysis Project” in the Kinect SDK documentation.

Tips and tricks

Efficient tagging

To tag efficiently, make sure you know all of the keyboard shortcuts for VGB; at the same time, the shortcuts that you’ll probably use most often are those in the following table.

Key stroke or combination	Action
Shift + Left Arrow / Shift + Right Arrow	Selects a range of frames to tag.
Enter	Sets the default maximum value.
Delete	Deletes the selected range or a single frame.
Ctrl + Left Arrow / Ctrl + Right Arrow	Moves the cursor to the previous or next frame.

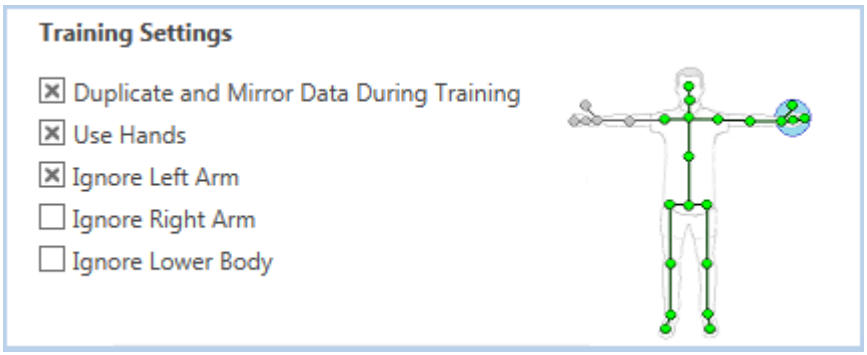
Key stroke or combination	Action
Page Up / Page Down	Selects the previous/next attribute in the **Tags** grid as the active attribute.

Some shortcut keys can also be combined. For example, by pressing Shift + Ctrl + Left Arrow/Right Arrow, you can quickly select multiple frames between tagged gestures, since Shift selects the frames and Ctrl jumps to the next/previous tagged gesture.

Use training settings

When used correctly, training settings can greatly improve the reliability and robustness of gesture detection. During the process of gesture creation, several training options are available in the creation dialog.

Figure 10. Shows the basic "Training Settings" that are available when creating a new gesture project.



Think about your gesture carefully. Which parts of the body are integral to the gesture? Which body parts would be better to ignore? By ignoring a body area, you provide the user with more flexibility and make your gesture more robust. For example, if your gesture depends mainly on the upper body, then selecting **Ignore Lower Body** will allow that gesture to be detected when the user is standing, seated, crouched, or when their legs are occluded by furniture, and so on.

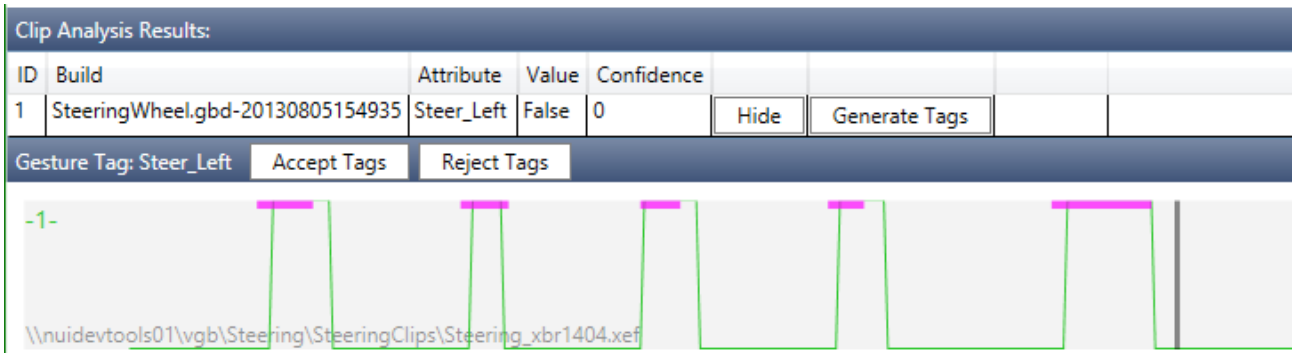
If you are confused about which training settings to use, try creating your gesture by using the Gesture Wizard (a link to this tool is available within the **Create New Project** dialog, located on the **File** menu). This tool will provide a step-by-step guide to help you learn more about the basic training properties and when to use them. You can modify existing settings in the gesture's property panel after creation.

Use 'Generate Tags'

After you have built a gesture database, you can start to use the 'Generate Tags' feature for help with tagging new clips to improve gesture detection. We recommend the following approach when using the 'Generate Tags' feature:

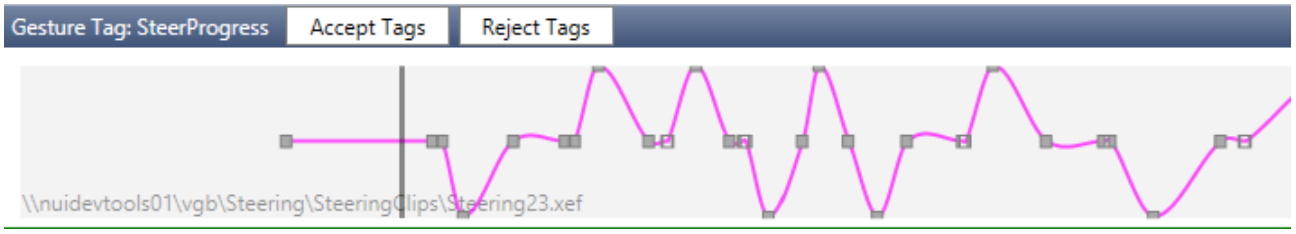
1. For discreet gestures (AdaBoostTrigger), follow these steps:
    a. Right-click on the analysis project and select 'Add Clip'
    b. Add a small number of clips to the project (10-20)
    c. Right-click on the analysis project and select 'Analyze'

d. Select the gesture database that you want to use for generating clips with (typically, this is the most recent database that you have trained).
e. For each untagged clip in the analysis folder, select the 'Generate Tags' button.
    i. Newly created tags will be colored pink in the timeline control.
    ii. Review the clips for accuracy and adjust the tags as needed (this usually requires fixing the first few frames at the start and end of each gesture).



f. After verifying that the tags are correct, select 'Accept Tags' or 'Reject Tags'.
    i. Accepted tags will convert from pink to blue, to show that they are final.
    ii. Rejected tags will be removed.
g. Compare the analysis results with the final tags within each clip. If the results are different, consider moving the clip into the training folder to help imrove detection. Try to maintain the training:testing ratio by moving 2/3 of the clips into the training folder and keeping 1/3 in the analysis folder.

2. For continuous gestures (RFRProgress), follow these steps:
    a. Ensure that each clip in the project has been tagged with one or more discreet gestures (for example, if you are tagging the continuous gesture, *JumpProgress*, then the discrete gesture, *Jump* must already be tagged).
    b. Right-click on the project and select 'Generate Tags' to open the 'Link Gestures' dialog.
        i. Select one or more discrete gestures (*Jump*) that can be mapped to the progress gesture (*JumpProgress*).
        ii. Specify start and end values for the continuous gesture which should correspond to the first and last frame of the discrete gesture.
        iii. Select 'Confirm' to generate the continuous gesture tags.
        iv. Review the clips for accuracy and adjust the tags as needed (this usually involves adding/adjusting mid-points).
    c. After verifying that the tags are correct, select 'Accept Tags' or 'Reject Tags'.
        i. Accepted tags will be converted from pink to blue, to show they are final.
        ii. Rejected tags will be removed.



Reduce training time

Depending on how much training data you have and the type of hardware your training computer has, training times for AdaBoost Trigger can potentially be long. There are, however, a few easy things that can be done to reduce training times.

Training a gesture database can be shared across multiple machines by having each machine build individual gesture projects with the command-line option **-build**. Once all projects are complete, they can be combined into a single database by using the command-line option **-join**.

AdaBoost Trigger has a project setting, Accuracy Level, that you can specify when configuring a project. This project setting controls the accuracy threshold that must be reached as a result of machine learning. Therefore, it affects how long training will take by stopping the search for weak classifiers when a certain error metric has been reached. For more information about Accuracy Level, see "AdaBoostTrigger Machine Learning" in the Kinect SDK documentation.

You can also exclude from training meaningless data (for example, a person standing idle for a long period of time) by marking all training frames as TRUE or FALSE. VGB clip files that have discrete gestures tagged only with TRUE values assume that all non-tagged frames are FALSE during training. However, if a clip file contains TRUE and FALSE values, then all frames not tagged will be excluded during training.

Another way to reduce training time and run-time cost is to mirror a gesture—for example, a right-handed golf swing could be used as a left-handed golf swing by mirroring the data. The **HorizontalMirror** property of **VisualGestureBuilderFrameSource** can be used to mirror the skeletal input data at run time. Essentially, you halve your training time by using this functionality, because you only have to train for one side, and at run time, you can use the same database for both sides or directions.

Command-line options

Since VGB turns the problem of writing gesture detectors into content building, you might want to build gestures overnight, on a daily basis, with other content. Options for building and analyzing gestures, joining or splitting gesture databases, and listing the contents of a database are available from the command line. These options can be used from the command prompt or in batch files and scripts; for more information see "VGB Command Line Options" in the Kinect SDK documentation.

Pros and cons

Visual Gesture Builder provides many benefits, but using it also has some costs. Consider the following pros and cons in assessing VGB for your project.

Pros

- Gesture detection is created and handled as content, rather than being a time-consuming engineering task.
- Gestures can be quickly prototyped and evaluated before any code is written.
- Non-engineers can create gestures—for example, designers, animators, technical artists and testers. No knowledge of machine learning, skeletal tracking, or programming is required.

- High accuracy for detecting gestures can be achieved—even in cases where skeletal data is very noisy, such as sideways poses.
- By tagging data appropriately, perceived latency can be made very low.
- The run-time costs to CPU and memory are low—on magnitudes of microseconds and kilobytes.
- The database size is independent of the amount of training data.
- There are very few thresholds to tweak and maintain.
- Live Preview (VgbView) enables fast iteration times for easy prototyping and testing.
- VGB provides a free, automatic test framework for gestures; that is, no extra engineering work must be done to test gestures, as in the traditional implementation of gesture detectors.
- A simple API is provided by the SDK.
- As tracking in the SDK improves, you can convert your clip files to work with the latest advances in depth and skeleton.

Cons

- It's time consuming to tag data—but at least it's not an engineering task, and therefore, you can outsource tagging.
- You require a powerful PC for training, but most developers already have a powerful PC for building other types of content.
- You require lots of disk space for storing raw (XRF) and processed (XEF) recordings, but luckily, hard disk drives are becoming cheaper.

Conclusion

Using traditional methods to create gesture detectors for Kinect is not a trivial task to do robustly. Visual Gesture Builder simplifies this task, which can make developers more productive and raise the quality of Kinect applications in terms of better gesture detection and reduced latency. Since tagging plays a huge role in obtaining good results, it is worth investing in a quality assurance team to approve data before adding the data to a training set.

References

Presentations

The following presentations are available from Download Center:

- *Innovative Solutions to Gesture Detection* – Claude Marais & David Quinn, Game Developer Conference 2012.
- *Gesture Detection using Machine Learning* – Claude Marais, Gamefest 2011.
- *Building Great Gesture Detection* – Claude Marais, Game Developer Conference 2011.
- *Xbox Studio Deep Dive: Designing, Debugging and Testing Your Kinect Title* – Claude Marais, Game Developer Conference 2011.