omek

Omek Beckon™ Motion Toolkit for Unity

Developer Guide

Legal Notice

Table of Contents

Introduction

The Omek Beckon™ SDK is a software package for multiple person motion tracking, using a single depth sensor. You can use the Beckon SDK to develop a Natural User Interface application, which reacts to player movements.

The Beckon Motion Toolkit for Unity is a package containing Unity components, prefabs, and sample scenes, which you can easily integrate into your Unity-based game application. The Motion Toolkit provides access to the major usage scenarios that the Beckon SDK supports, enhanced for Unity developers. These include:

- Controlling Unity GameObject positions
- Displaying players as images, icons or animated avatars
- Determining how people are selected as players
- Enabling players to control cursors via hand movements

This guide is intended for programmers who are familiar with game development in general and the Unity development environment in particular.

Additional Resources

The Motion Toolkit is based on another Omek package called the Omek Beckon C# Extension. This guide provides a brief overview of the Motion Toolkit components and how to use them. For more detailed explanations of Beckon SDK options, especially those related to player management, please refer to the Beckon C# Extension Developer Guide, which you can find at the Omek Support Site.

You can also refer to the following link, which is a forum for Unity developers using Beckon.

Note:
Although you don't have to install the Omek Beckon SDK in order to use the Motion Toolkit, we recommend installing it to provide easy access to sample code, documentation and gesture packages. You can download the Beckon SDK from the Omek site.

Getting Started

Importing the Motion Toolkit Package

To start using the Motion Toolkit in your Unity project:

1. Open your project in the Unity environment.
2. Import the **MotionToolkit** package into your project. The **MotionToolkit** project appears in your Project pane.
3. From the Prefabs folder, add a **UnityBeckonManager** prefab to your scene.

Note:
If your game contains multiple scenes, create* UnityBeckonManager *only once, and call the Unity function DontDestroyOnLoad on this GameObject.

These are the folders you will see included in the **MotionToolkit** project:

- **Editor** - contains editor scripts for displaying various components in the Unity Inspector.
- **Prefabs** - contains pre-configured prefabs of the major Toolkit components.
- **SampleAssets** - contains sample scenes and some related assets.
- **Scripts** - contains various Toolkit scripts.
- **Textures** - contains textures used as defaults by the Toolkit scripts.
- **Plugins** - contains dll files that the Toolkit uses.
- **Resources –** contains a sample gesture pack.

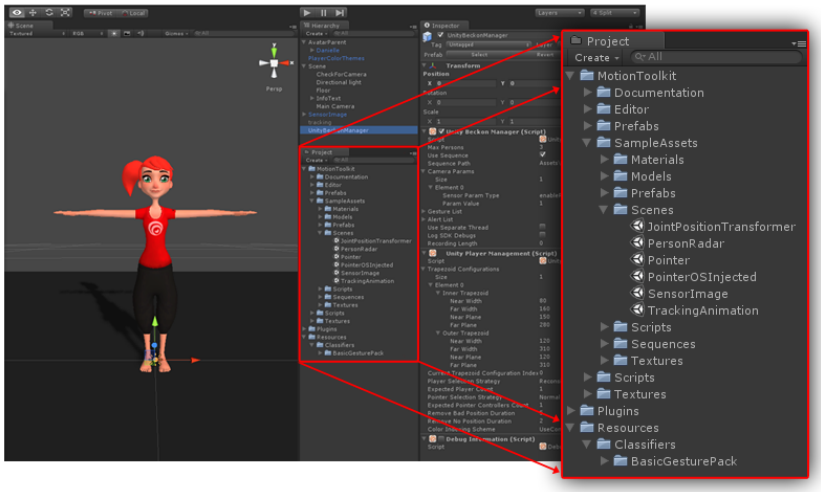

Figure 1: Unity IDE with Motion Toolkit Components

You can access all of the Motion Toolkit components under the **Component -> Omek** menu. Many of them are also available as pre-configured prefabs in the Prefabs folder, in the project pane.

Note:
When building an application that uses the Motion Toolkit, you must configure Unity to be compatible with .NET 2.0.

If the API Compatibility Level is not set correctly the following error will be received: "ArgumentException: The Assembly System.Web is referenced by log4net. But the dll is not allowed to be included or could not be found."

To set the Unity compatibility level to .NET 2.0:

1. Go to **Edit -> Project Settings -> Player**.
2. Choose the Standalone platform setting.
3. In Other Settings, set the API Compatibility Level to ".NET 2.0".

Note:
When deploying a Motion-Toolkit-based application, copy all dlls from the plugin folder and the Resources/Classifiers folder to the same directory as the game executable.

Exploring the Motion Toolkit Options

Now you're ready to start exploring the Motion Toolkit options. The easiest way to do this is by running the sample scenes included in the Toolkit (which you can find in the SampleAssets/Scenes folder), reading their code, and reading the sections of this guide that refer to the components used in each scene.

The next section describes the sample scenes included in the Toolkit.

Motion Toolkit Sample Scenes

The following table summarizes the sample scenes included in the Motion Toolkit.

Scene	Description
JointPositionTransformer	This scene shows how to control the position of a GameObject according to the real-world movements of a person.
TrackingAnimation	This scene shows how to animate an avatar that mimics person movements in real-time.
PersonRadar	This scene shows a top-down view of the playground, where each person is represented by an icon.
SensorImage	This scene shows how to display the image captured from the 3D sensor, including options to display people as color images or depth images.
Cursor	This scene shows how to transform persons' hand movements to on-screen cursor movements, which can activate GUI items such as buttons. It also shows how to listen to gestures.
CursorOSInjected	This scene shows how to control the Windows cursor according to a person's hand movements.

Configuring the Motion Toolkit

The Motion Toolkit supports several configuration options, which determine how the underlying Beckon SDK handles visual input, thread architecture, and player management. The following table summarizes the most important configuration options.

UnityBeckonManager component parameters	
Parameter	**Description**
Use Sequence; Sequence Path	By default, the Beckon SDK is configured to receive real-time data from a depth sensor. You can also choose to supply a pre-recorded video sequence as input, by setting the **Use Sequence** option and the desired **Sequence Path**.
Use Run Sensor	If you set this parameter to "true", the Beckon SDK's processing loop runs in a separate thread, which may be advisable for optimizing performance. Otherwise, it will run on the main Unity thread.
Max Persons	Determines the maximum number of persons in the scene that Beckon will track.

UnityPlayerManagement component parameters	
Parameter	**Description**
Player Selection Strategy	Determines how players are selected from among the tracked persons. *See the Beckon C# Extension Developer Guide for details.*
Expected Player Count	Determines the maximal number of players that Beckon will manage.
Cursor Selection Strategy	Determines how players are selected to control cursors. *See the Beckon C# Extension Developer Guide for details.*
Expected Cursor Controllers Count	Determines the maximal number of cursor controllers that Beckon will manage.

Trapezoid Configurations	Contains parameters related to monitoring of the players' locations. You can define different trapezoid-shaped areas on the playground, for different states in the game (single-player, multi-player, menu), and switch among them during the game.
Current Trapezoid Configuration Index	Determines the active trapezoid area that Beckon will use to track persons.

Trapezoid Configurations component parameters

Parameter	Description
Inner Trapezoid	A trapezoid-shaped area on the playing floor that is considered a valid location for players.
Outer Trapezoid	A trapezoid-shaped area on the playing floor, larger than (and containing) the **Inner Trapezoid**. If a player is outside of the **Outer Trapezoid**, he/she cannot be tracked. The area between the outer borders of two trapezoids is considered a "warning" zone. *See the Beckon C# Extension Developer Guide for details.*

Note:
The Motion Toolkit components contain additional configuration options, which you can learn about by reading their tooltips.

Controlling GameObject Positions

The Toolkit provides two components for mapping between the position of a person's joint to the position of a GameObject inside the game world:

- **JointPositionTransformer** - maps the 3D real-world position of a person's joint to a GameObject local position in a 3D area in the game world.
- **JointImagePositionTransformer** - maps the 2D image position of a person's joint to a GameObject local position in a 2D area on the screen.

To use one of these components to control your GameObject's position, add the component to your GameObject.

When using **JointPositionTransformer**, you will need to define two cubic areas:

- **WorldBox** - the 3D area in the real world in which the person joint is tracked. The area is defined in centimeters, in terms of distance from the sensor. If the person moves outside of the world box along a certain axis, the displayed position on that axis is "clamped" to the box boundaries.
- **TargetBox** – the 3D area in the scene to which the joint position is mapped. The area is defined in Unity units (usually meters). The ratio between the world box and the target box determines the scaling that is performed.

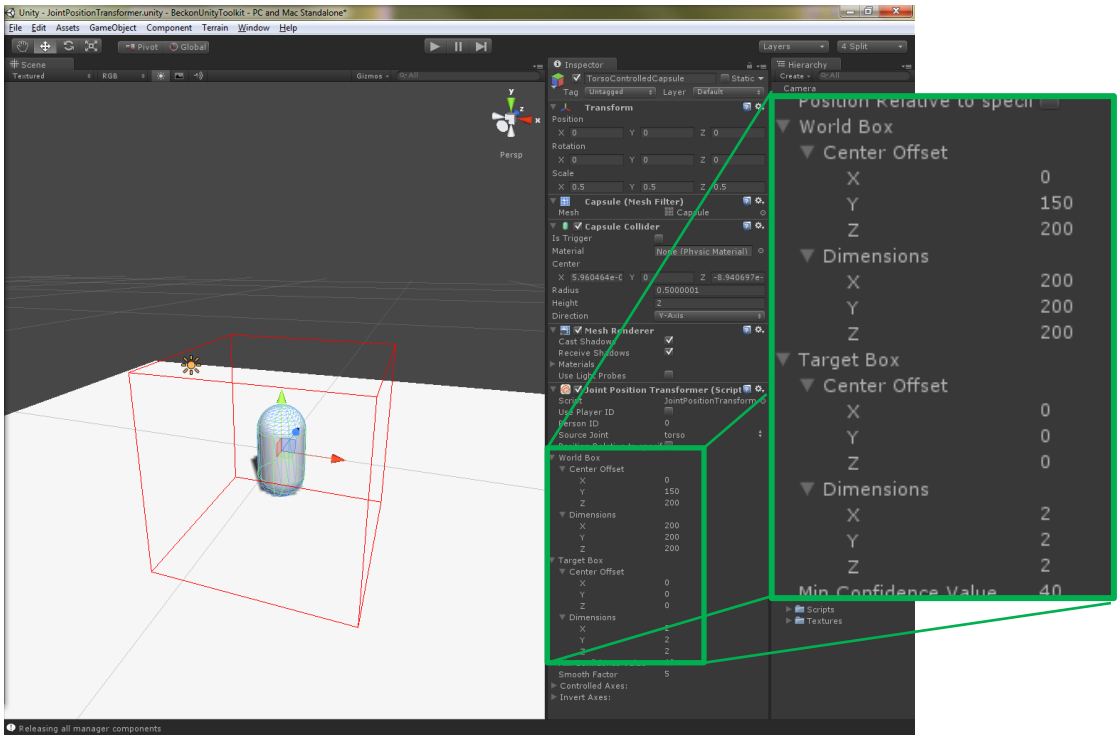

Figure 2: JointPositionTransformer Component with its Target Box

JointImagePositionTransformer operates very similarly to **JointPositionTransformer**, except that instead of defining 3D areas, you define 2D areas in its **ImageRect** and **TargetRect** properties. **JointImagePositionTransformer** doesn't affect the GameObject Z coordinates. You can use **JointImagePositionTransformer** to overlay a texture on the image captured from the depth sensor, and map its position to the position of a specific person's joint.

Other parameters that control joint mapping and display are:

Parameter	Description
Use Player ID	If set to "true", the **PlayerID** parameter determines the player whose joint is tracked. If set to "false", the **PersonID** parameter determines the person whose joint is tracked.
Person ID / Player ID	The ID of the person or player whose joint will be mapped.
Source Joint	The specific joint to map to a GameObject.
Position Relative to Specific Joint	If set to "true", the mapped joint's position will be calculated relative to another specific joint. For example, the hand joint position may be taken relative to the torso joint, causing changes in the person's body location to be ignored, and reflecting only the hand movements in relation to the torso.
Relative To Joint	The joint relative to which the mapped joint's movements are calculated.
Controlled Axes	Sets the axes (X/Y/Z) along which motion will be tracked. Movements along other axes are ignored. For example, if you set **Controlled Axes** to "X, Y", movements along the X and Y axes will be tracked, but movements along the Z axis will be ignored.
Invert Axes	Sets the axes (X/Y/Z) along which motion will be inverted. For example, if you set **Invert Axes** to X, right-to-left movement will be mapped to left-to-right movement.

Parameter	Description
Min Confidence Value	Beckon produces a confidence value for tracked joint positions, reflecting the estimated accuracy of the joint's position data. If the mapped joint's position has a confidence value which is lower than the value set in this parameter, the new position will be ignored, and the most recent position with a higher confidence value will be retained.
Smooth Factor	Determines the degree of the "smoothing" that Beckon applies to joint motions. The smaller the **Smooth Factor** value, the smoother the motion will be. (Stronger smoothing also introduces a larger delay in display). To disable smoothing, set this parameter to 0.

Basic Person Display

In your application, you can display persons that Beckon tracks in one of two basic modes:

- A two-dimensional image corresponding to the sensor image
- An icon in overhead view

This section describes how to implement each of these display types.

You can also display a person as an animated avatar, which also allows the person to interact with your game (see *Avatar Animation*).

Displaying Person Images

To display players as images, which are similar to those captured by the depth sensor, add a **SensorImageConfigurator** component or prefab to your scene.

You can then set the Unity RenderType parameter to one of the following values:

- **OnGUI** – shows a simple two-dimensional image. (This mode has the worst performance and will usually not be used in operational code.)
- **GUITexture** - shows a simple two-dimensional image.
- **Plane** – supports advanced options, such as using materials and shaders, or displaying the image on a two-dimensional surface which can be rotated, but requires more work to configure.

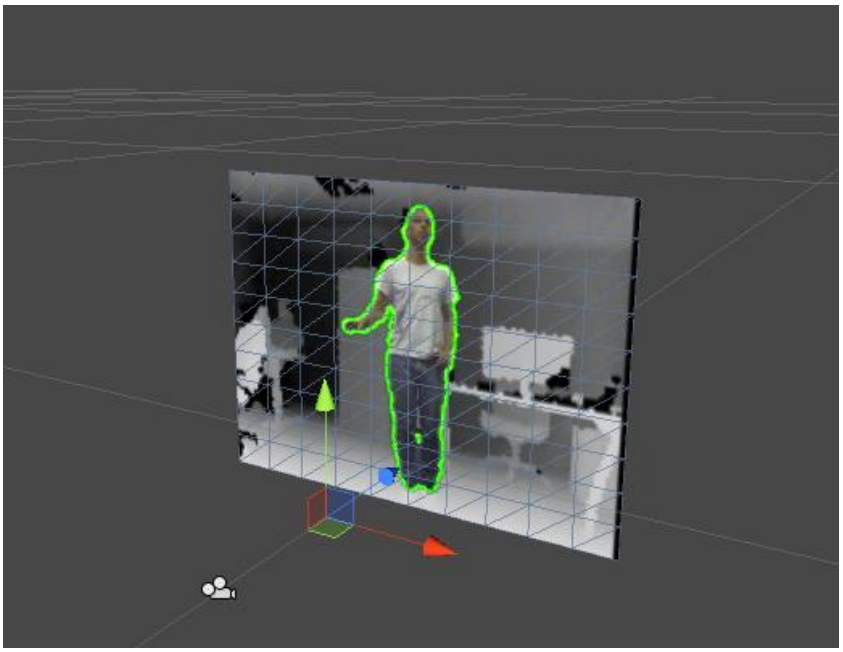

Figure 3: Plane Rendering of a Person Image

You can choose to display the scene's background as one of the following:

- The depth image captured from the sensor
- The color image captured from the sensor (if this is supported by the sensor, and enabled by using the **CameraParams** field in **UnityBeckonManager**)
- No background image

The person image can be displayed as:

- The depth image captured from the sensor
- The color image captured from the sensor (if enabled)
- A silhouette filled with a solid color

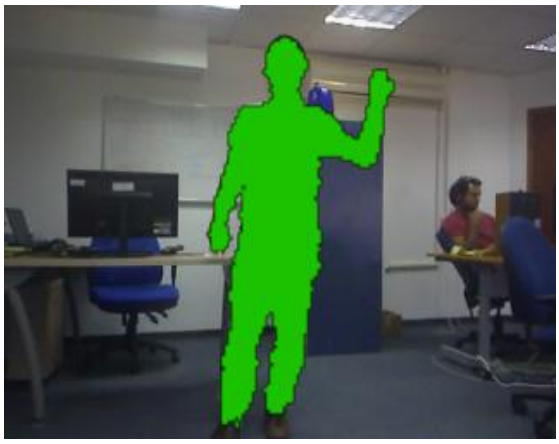

Figure 4: Color Background and Silhouette Image

Figure 5: Depth Background and Color Image

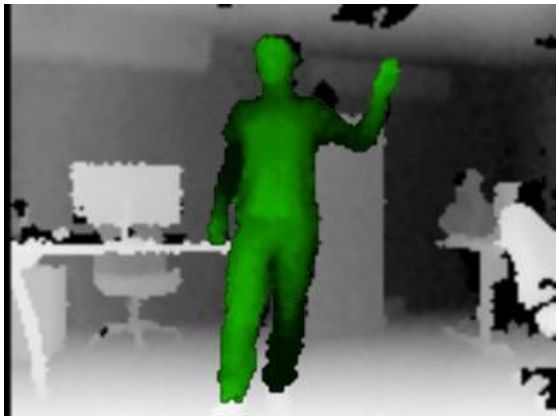

Figure 6: Depth Background and Depth Image

Figure 7: No Background and Color Image

Displaying Person Icons

There are two Motion Toolkit components that allow you to display persons as icons in an overhead view:

- **SimplePersonRadar** - maps the position of each person to a rectangular area on the screen, and displays an icon for each person. **SimplePersonRadar** has a **World Coordinate** field that defines the area on a X/Z plane to map to the screen.
- **TrapezoidPersonRadar** - performs a very similar task, but maps the area defined in **InnerTrapezoid** of **UnityPlayerManagment** to a trapezoid on the screen, defined in pixels in the **Screen Trapezoid** property.

Both components have **ScreenCoordinates** and **SnapTo** fields to define their placement on screen. The Toolkit uses the same icon to display all persons, but with different colors according to their player state.

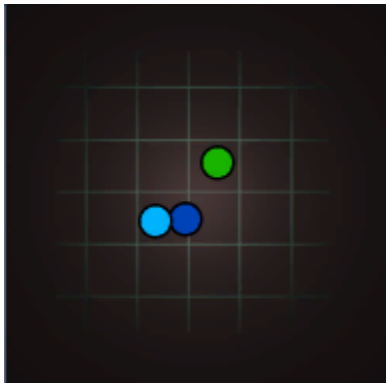

Figure 8: Simple Person Radar

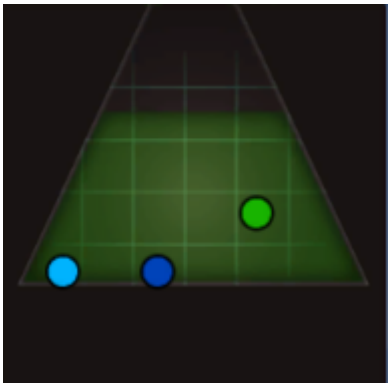

Figure 9: Trapezoid Person Radar

These are the parameters that control person icon display:

Parameter	Description
Bg Texture	An image to display in the background.
Person Texture	An icon to display for all persons.
Person Rect Size	The size of the icon to display.
Person To Show	An optional list of IDs of persons to display as icons. If the list is empty, all persons will be displayed.
Show Non Player	If set to “true”, all persons will be displayed as icons. If set to “false”, only players selected by **UnityPlayerManagment** will be displayed as icons.
Radar Color	Defines colors for displaying each person state (see *Specifying Colors for Person and Cursor Display*).

Specifying Colors for Person and Cursor Display

The Toolkit allows you to set display colors for person images, person icons and cursors.

You may want to set different colors for persons with different “player states”.
The player states are:

- **Non-player** – a person who has not been selected as a player.
- **Player** – a person who has been selected as a player.
- **Cursor controller** – a person who is controlling a cursor.

The toolkit provides two convenient ways to do this, either by setting colors in each component separately, or by setting them globally in the **PlayerColorThemes** component.

Setting Colors by Component

You can set the Colors field of each component that displays persons and cursors. In this field, you can define a different color for each player state. The colors you define in a component will apply only to persons, icons and cursors displayed within that component.

For example, if your scene displays person images, cursors and a top-down icon display of the persons in view, you can set the colors of the images in the Colors field of SensorImageConfigurator, the colors of the person icons in the Radar Colors field of the Radar component, and cursor colors in the PointerVisualization component.

Setting Colors Globally Using PlayerColorThemes

You can set display colors globally, by adding a **PlayerColorThemes** component or prefab to the scene and setting the colors in this component. The colors set in **PlayerColorThemes** for persons, icons and cursors will override these color settings in other components. **PlayerColorThemes** also has some custom color fields, which you can use to set colors for other types of displayed objects.

Since there may be several persons in each player state, **PlayerColorThemes** allows you to define several colors for each person within each state. A common usage scenario is to define different shades of the same basic color for each state. For instance, you can define different shades of blue to display players, and different shades of green to display cursor controllers. This allows you to differentiate among player states, while still differentiating among the different persons.

The following figure is one example of how to define colors for person, icon and cursor display in **PlayerColorThemes**. Note that in the field names, “Mask” refers to a person image, and “Rader” refers to a person icon.

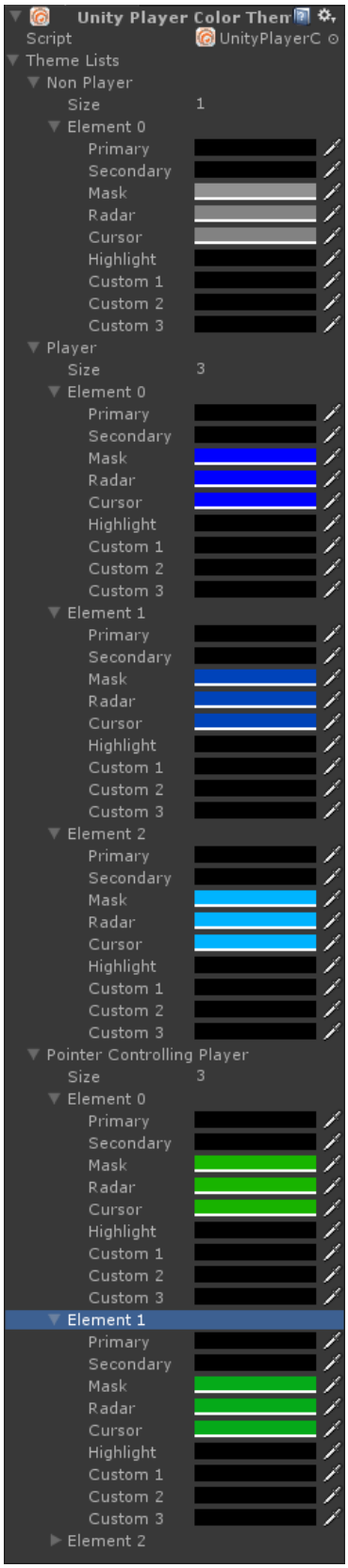

Figure 10: Setting Colors in PlayerColorThemes

Avatar Animation

By using the Toolkit's **SkeletonTrackingAnimation** component, you can easily map a person's movements to avatar movements using the standard Unity animation system. This allows the person to interact with your game via body movements.

To map a person to an avatar:

1. Add the **SkeletonTrackingAnimation** component to a GameObject that also contains an Animation component.
2. Specify the Transform for each joint in the component inspector window. If your model uses a standard naming convention (such as those created by 3D Max or MotionBuilder), you can use the Configure Automatically button to let the script find the relevant Transform for each joint.
3. Specify which person (or player) to map to the avatar, using the **PersonID** (or **PlayerID**) field.

Upon creation, the script creates a dummy AnimationState called **trackingAnimation**. You can use this AnimationState to control blending with other animations. You can start, stop or cross-fade the tracking animation, as you do with any other kind of animation.

Note:
By using* SkeletonTrackingAnimation*, you can only animate avatars in place. If you want to move them around in the scene, use* JointPositionTransformer *in addition (see* Controlling GameObject Positions *for more information, and examine the* TrackingAnimation *scene for an example of how to do this).

We recommend defining a default animation that is set to play automatically. This is so that whenever there is no tracking information (for instance, when a person is blocked from view), the avatar will return to this default animation.

Note:
The avatar must initially be placed in the scene in a T-pose, so that Beckon can calculate its dimensions.

Figure 11: Avatar in T-Pose

These are the parameters that control **SkeletonTrackingAnimation**:

Parameter	Description
Use Player ID	If set to “true”, the **PlayerID** field determines which person controls the avatar. Otherwise, the **PersonID** field determines which person controls the avatar.
PersonID / PlayerID	The ID of the person or player that controls the avatar.
Default Layer Of Tracking Animation	The animation layer that is assigned to the “trackingAnimation” AnimationState by default. This may be changed later by directly accessing the AnimationState.
Start Animation Automatically	If set to “true”, the tracking animation will be played automatically upon creation of the avatar.
Use Smoothing	If set to “true”, a smoothing algorithm will be applied to the avatar’s motion.
Animation Smoothing	The degree of smoothing to apply to the avatar’s motion (lower values produce smoother but somewhat delayed motion).
Mirror	If set to “true”, the avatar’s movement will be inverted from right to left in relation to the person’s movement. This causes the display to behave like a mirror image, as opposed to displaying the motion as the sensor “sees” it.
Use Retarget Skeleton	If set to “true”, the Beckon **Retarget Skeleton** feature for mapping person skeletons to avatars is used (recommended).
Confidence Change Rate	Determines how fast to fade movement out when Beckon loses tracking of a limb (for example, when an arm is moved behind a person’s back). When Beckon fades out movement for a joint, Beckon no longer affects that joint’s animation, and lower-level animations (such as those defined in Unity) take affect.
Confidence Change Safety Time	Determines how long to wait after a joint loses confidence, before Beckon starts to fade its movement out.

Cursor and GUI Buttons

Adding Cursor

The **UnityCursorManager** component provides an easy way to use a person's hand movements to control a cursor. See the *Beckon C# Extension Developer Guide* for details about how to select players to control a cursor.

The Toolkit provides two options for player-controlled cursors:

- If you need only one cursor, you can override the operating system cursor by selecting **UnityCursorManager**'s **Override OS Cursor** option.
- If you want to implement multiple cursors, you will need to write your own code to interface with multiple cursors, since Unity supports only one cursor. The **CursorVisualization** script demonstrates how to manage multiple cursors, and change their color according to the person controlling each one. (See *Setting Colors Globally* for information on how to use **PlayerColorThemes** instead).

Adding Button

In the Toolkit's **Cursor** scene, you can see an example of how an application reacts to cursor actions. The **HotSpotButton** is clicked when a cursor hovers over the button. The **GestureButton** is clicked when a person performs a specific gesture. Both buttons use **CursorListener**, which monitors an area on screen and uses SendMessage to notify other scripts about its current status.

Note:
The cursor coordinate values that* UnityCursorManager *provides are in the range [0,1]. Examine the* CursorListener *and* CursorVisualization *components to see how these are mapped to screen coordinates.

Appendix A: Player Image Components in Detail

This appendix describes the player image components used in the **SensorImage** scene. You may want to use the player image components directly, for additional flexibility.

The basic player image components used in the **SensorImage** scene are **PlayerMask** and **DepthCamImage**. You can add these components to any GameObject that has a renderer or a GUITexture component. The player image components will set the main texture of the renderer or the GUITexture component to an image captured from the depth sensor.

DepthCamImage will set the texture to the depth or color image captured from the sensor, while **PlayerMask** will set the texture to a cutout image of a specific person. Using these components, you can display the captured images on any model in the scene.

The **MultiplayerMasks** script receives a pre-configured prefab containing a **PlayerMask**, and materials and colors to be used for display of different player states (NonPlayer, Player or CursorControlling). The script automatically creates a copy of the prefab with the relevant parameters for each person in the scene.

In a real game it might be more effective to use this script directly instead of using **SensorImageConfigurator**, as it allows more flexibility, and in some cases, better performance.