VIRMEN MANUAL

by Dmitriy Aronov

Last updated: November 27, 2012

TABLE OF CONTENTS

INTRODUCTION

Welcome to ViRMEn – the **Vi**rtual **R**eality **M**atlab **En**gine. ViRMEn is a Matlab-based software package that contains 3 components:

1. A graphical user interface (GUI) for interactively designing virtual environments (worlds).
2. An engine that performs 3D rendering of virtual worlds on a computer monitor or projector.
3. An object-oriented toolbox for manipulating virtual worlds programmatically outside of the GUI.

ViRMEn achieves fast graphics by directly accessing low-level OpenGL commands instead of using built-in Matlab functions. The engine also recognizes physical boundaries in virtual worlds and performs continuous-time collision detection (CTCD) to manage the animal's contact with these boundaries.

ViRMEn is designed to be highly customizable. The engine can apply arbitrary spatial transformations to virtual worlds to display them on practically any type of screen (flat, conical, toroidal, etc.). The engine can also access any type of an input device (such as an optical mouse) to obtain the animal's velocity information. Finally, the engine runs custom user-written Matlab code that can implement experiment logic, reward delivery, data logging, etc.

INSTALLATION

SYSTEM REQUIREMENTS

- **Windows**: Because ViRMEn uses .NET libraries, it will only run on a Windows computer. ViRMEn has run successfully on computers with Windows XP and 7. ViRMEn will run fine on a 32-bit or a 64-bit computer.
- **Matlab**: One of the later versions is required, should support class programming and .NET libraries. ViRMEn has run successfully in versions 2010 and later. If you plan to interface with a NI-DAQ card, you may want to install 32-bit Matlab because of the differences in the Data Acquisition Toolbox between 32 and 64-bit versions (see Matlab help). 32-bit Matlab can be installed on a 64-bit machine.
- **C compiler**: A compiler may be required for using ViRMEn mex files. Any compiler (such as Visual Studio) should work. Install the software on your computer and run >> mex -setup in Matlab to define it as the default compiler for Matlab.

INSTALLATION STEPS

- Place the virmen folder anywhere on your computer.
- Add the virmen folder, *with subfolders*, to the Matlab path.
- Run >> virmen at the command line in Matlab.

VIRMEN FOLDERS

The ViRMEn folder contains the following subfolders.

- **bin**: contains all files related to the engine, the GUI and the documentation. Nothing in this folder should ever be changed by the user.
- **defaults**: contains files that store various user preferences for the GUI. See customizing ViRMEn.
- **experiments**: contains experiments created by the user. Each experiment is defined by a .mat file and a .m file. The .mat file contains a variable exper, which stores the experiment itself. The .m file contains all custom Matlab code used by the experiment.
- **movements**: contains all functions that provide velocity input for movement in the virtual world. See movement functions.
- **objects**: contains all definitions of 3D objects recognized by ViRMEn. See object types.
- **shapes**: contains all definitions of 2D shapes recognized by ViRMEn that are used to create textures for objects. See shape types.
- **transformations**: contains all transformation functions that can be used to transform a 3D representation of a world into a 2D projection onto a virtual reality screen. See transformation functions.

UPDATING VIRMEN

You can check the date of latest update of ViRMEn by pressing the *About ViRMEn* button in the GUI. A window that pops up will also allow to download the latest version. You can also download the latest version directly from the password-protected site http://tiny.cc/virmen. To update ViRMEn to a newer version, only update the "**bin**" subfolder. Other subfolders may contain files specific to your experiments, which should not be overwritten.

TROUBLESHOOTING

Problem	**Try solution**

Error after installation:
`Could not load file or assembly 'OpenTK.dll' or one of its dependencies.`
	To enable MATLAB to load the assembly, try updating the file
	
	$MATLABROOT\bin\[win32|win64]\matlab.exe.config:
	You will need to add the following configuration options:
	
	`<configuration>`
	`  <runtime>`
	`  <loadFromRemoteSources enabled="true"/>`
	`  </runtime>`
	`</configuration>`
	
	As there already should be some content in this file, the resulting file should look like:
	
	`<configuration>`
	`  <startup useLegacyV2RuntimeActivationPolicy="true">`
	`  <supportedRuntime version="v4.0"/>`
	`  <supportedRuntime version="v2.0.50727"/>`
	`  </startup>`
	`  <runtime>`
	`  <loadFromRemoteSources enabled="true"/>`
	`  </runtime>`
	`</configuration>`

GRAPHICAL USER INTERFACE (GUI)

The following sections will take you through a series of steps to acquaint you with the ViRMEn graphical user interface (GUI). At this point, we will only be creating worlds, with no coding. The next section of runtime programming will begin to add custom code to your experiments.

Unless otherwise indicated, all buttons and dropdown menus you will be asked to press are located on the toolbar of the GUI. Not all buttons are available at all times – make sure the correct window in the GUI is activated first by clicking on it.

VIRMEN GUI BASICS

STARTING VIRMEN

- Run >> virmen at the command line in Matlab. The ViRMEn GUI will start. It may take a while the first time because Matlab has to load custom ViRMEn classes, but should start faster on subsequent runs.

CHANGING LAYOUTS

- The GUI will start in the Experiment layout, which is used to edit general properties of your experiment. The layout contains three windows: Custom variables, Experiment properties and Worlds menu. These will all be explained later. One of the windows is active and is indicated by a red border. Click inside different windows to make them active instead.
- Press the *World layout* button. The GUI will switch to the World layout, which contains three different windows. This layout is used to place objects into your world and arrange them in virtual space.
- Press the *Texture layout* button. The GUI will switch to the Texture layout and display three more windows. This layout is used to edit the textures of the objects in your worlds.
- Press the *Experiment layout* button. You are now back in the Experiment layout.
- Experiment, World and Texture are the three built-in layouts of the ViRMEn GUI. You can, however, change the arrangement of windows in each of these layouts. You can even create additional layouts that combine windows in the way you like. See customizing layouts.

UNDO AND REDO

- You can undo or redo any changes you will make to your experiment from now on. The GUI stores in history the last 10 changes you've made. Press the *Undo* or *Redo* buttons.

HELP

- Press the *ViRMEn manual* button to open this manual.

ENVIRONMENT (WORLD) MANIPULATIONS

ADDING AND DELETING WORLDS

- Go the Experiment layout of the GUI and select the Worlds menu window by clicking on it. The Worlds menu shows thumbnails of all environments (called "worlds") used by your experiment. If you just started, your experiment contains a single world, which is blank except for a square-and-arrow symbol. This symbol shows the location where the animal is initially placed when the world is loaded.
- To add more worlds, press the *Add world* button. More blank worlds will appear in the worlds menu.
- To delete a world, press on the thumbnail to select it. Then press the *Delete world* button.

NAMING A WORLD

- Whenever you add a new world to your experiment, it is automatically given the name virmenWorld. When multiple worlds get the same name, they are indexed virmenWorld(1), virmenWorld(2), etc. You may wish to give some worlds custom names. To do so, click on the world thumbnail to select it and then press the *Rename world* button. You can also click on the world name above the thumbnail. Enter the new world name, which must be a valid Matlab variable name.

IMPORTING OR DUPLICATING A WORLD

- You can import an entire world that you've previously designed into the current experiment. Press the *Import world* button. A window will open, giving you the list of all previously created experiment names.
- Click on an experiment name. The window will display thumbnails of all worlds in that experiment. Click on a thumbnail to import that world into the current experiment.
- Note that the list includes the current experiment. Importing a world from the current experiment will duplicate it. This is useful for designing multiple worlds that are largely the same without having to repeat the entire design process.

OBJECT MANIPULATIONS

DRAWING AN OBJECT

- You are now ready to design a world. To do so, select the world you want to change in the Worlds menu and press the *World layout* button. Alternatively, you can press Edit under the thumbnail of the world.
- Click on the World sketch window, which contains a 2D sketch of your world. If your world is blank, it will only contain a square-and-arrow symbol indicating the starting location of the animal. The window will become active, as indicated by a red border.
- In the *Add object* dropdown menu, select the object type you wish to add. Let's start with a VerticalCylinder. On the world sketch, click locations where you wish to add cylinders. Let's start with 2 locations. When you're done adding locations, double-click or press Enter.

- The World sketch window shows a top view of your world. If you've added two cylinders, the world sketch shows two circles.
- Note that the cylinders do not appear in the World drawing menu, which is supposed to show the 3D view of your world. This is because your object does not yet have a texture and is therefore invisible.
- Note that the two cylinders you've added are treated as a single object that happens to have multiple locations. This means that any properties you change, such as the radius or texture, will apply to both cylinders. If you wish to have two cylinders that are different, you must add two separate objects containing one location each.
- In addition to using object types available in the *Add object* dropdown menu, you can create your own object types. See custom object types.

ZOOMING IN AND OUT

- You can zoom in on a part of the world by left-clicking in the World sketch window and dragging a rectangle over the region of interest.
- To zoom out, right-click within the World sketch window.
- To return axes to the default x and y limits, hold the Shift key and left-click in the World sketch window.
- You can change the default x and y limits of the world sketch by editing the default properties file.

SELECTING AN OBJECT

- You can select an object in one of two ways: 1) By clicking on it in the World sketch window or 2) by choosing it from the dropdown menu in the Object properties window. The selected object is colored red on the world sketch.
- You can select the start location of the animal by clicking on the square-and-arrow symbol or by choosing "Start location" from the dropdown menu. This will allow you to change the animal's starting location and view angle.

NAMING AN OBJECT AND ITS TEXTURE

- New objects are given default names according to their type, such objectVerticalCylinder. Objects that have the same name are indexed, such as objectVerticalCylinder(1), objectVerticalCylinder(2), etc. The objects' textures are similarly named virmenTexture(1), virmenTexture(2), etc. For programming purposes, it is good to give meaningful names to your objects. To rename an object and/or its texture, select it and press the *Rename object* button (or the *Rename* button next to the dropdown list or the name of the texture above its image). The name must be a valid Matlab variable name.

DELETING AN OBJECT

- To delete an object, select it and press the *Delete object* button.

SORTING OBJECTS

- ViRMEn lists objects in the order in which they were created. To re-organize object names into any order you wish, press the *Sort objects* button. A window will pop up, allowing you to move individual object names up or down, or to sort object names alphabetically.

CHANGING OBJECT PROPERTIES

- Each object type has properties associated with it. The properties are listed in a table in the Object properties window and can be modified after the object has been created. For example, try changing the radius of the cylinders you've created.

CHANGING THE ANIMAL'S STARTING LOCATION

- The animal's starting location (indicated by the square-and-arrow symbol) is treated as an object has four properties associated with it: X, Y, Z, and rotation. These are the starting location and view angle of the animal in the world (i.e., where the animal is placed when the world is loaded during an experiment). You can edit these properties by selecting the animal's location and changing values in the Object properties table. Note that different worlds in the same experiment can have different starting locations.
- By default, the location is set to (0, 0, 0) and the rotation is set to 0. View angle is offset by $\pi/2$ radians from the rotation value, such that the rotation of 0 corresponds to facing "north." You can change the default location and rotation by editing the default properties file.

CHANGING OBJECT LOCATIONS

- In addition to other properties, each object has locations associated with it. These locations can have different meanings for different object types. For instance, for cylinders each location indicates an instance of a cylinder (e.g., two locations = two cylinders). For a wall, locations indicate endpoints and corners (e.g., four locations = single wall with two corners).
- Locations are listed in a table in the Object properties window. You can manually modify individual locations by changing values in this table.

ADDING AND DELETING OBJECT LOCATIONS

- To add a location, make sure that the Object properties window is active by clicking on it. Then press the *Add object location(s)* button. A new row will be added to the table of locations.
- To delete a location, select the row(s) of the table you wish to delete. Then press the *Delete object location(s)* button.

ENTERING A SYMBOLIC LOCATION FOR AN OBJECT

- Suppose you want to create a row of 9 cylinders, spaced at 25 units from -100 to 100 along the x-axis. Rather than entering each location manually, you can enter a Matlab expression to create cylinders in bulk. To do so, press the *Symbolic object location(s)* button (or the *Symbolic expression* button next to the table of locations). Enter -100:25:100 in the first box; this will generate x values from -100 to 100.

You can enter an array of the same size into the second box, but since all of the y values for the 9 cylinders are the same, simply enter 0.

IMPORTING OR DUPLICATING AN OBJECT

- You can import an object that you've previously created. To do so, press the *Import object* button. A window will appear, giving you the list of all previously created experiment names.
- Select the experiment from which you wish to import an object. Images of all available objects will appear as thumbnails. Click on the thumbnail of the object you wish to import.
- Click on the location(s) where you wish to place the imported object. When you're done, double-click or press Enter.
- Note that the list includes the current experiment. Importing an object from the current experiment will duplicate it.

TEXTURE MANIPULATIONS

DRAWING A SHAPE

- Now that you've created at least one object, you can create a texture for it. Select the object and press the *Texture layout* button (you can also double-clicking the object or click the texture image).
- If you're working with a newly created object, you will see a blank square in the Texture sketch window. This is the boundary of the texture. The Texture drawing window shows the triangulation of your texture. A blank texture is divided into two triangles.
- Activate the Texture sketch window by clicking on it. In the *Add shape* dropdown menu, select the shape type you wish to add. Let's start with a Rectangle. On the texture sketch, draw the rectangle somewhere in the middle of the texture.
- Note that the rectangle shape you've drawn does not change the triangulation of your texture. This is because you haven't yet computed the texture.
- In addition to using shape types available in the *Add shape* dropdown menu, you can create your own shape types. See custom shape types.

ADDING A COLOR TO A TEXTURE

- By drawing a rectangle, you've divided the texture into two regions: the inside and the outside of the rectangle. Each region needs to be colored.
- To add a color to one of the regions, make sure the Texture sketch window is activated. Then click the *Add color* button. In the Texture sketch window, click inside all regions where you wish to add a new color. When you're done, double-click or press Enter.
- A menu will ask you to choose a color for the regions you selected. Note that if you clicked inside several regions, all of those regions must now have the same color. To assign different colors to different regions, you must add colors separately.

CHANGING COLOR OR TRANSPARENCY

- Once a color has been added to a texture, you can change it by selecting it and modifying the RGB values in the Shape properties table. You can also press the *Set color* button or double-click on the color to choose a color interactively.
- To change the transparency value of a color, modify the Alpha property in the Shape properties table. By default, all Alpha values are set to NaN, meaning that transparency is disabled, and all the colors are completely opaque. You can set the Alpha value to anything from 0 (completely transparent) to 1 (completely opaque).
- Transparency processing slows down the ViRMEn engine. If all of your objects are opaque (and will remain opaque throughout the experiment), set all transparency values on all textures to NaN. If even one color in the world has a numeric Alpha value from 0 to 1 assigned to it, transparency processing will be enabled. This is true even if all values are set to 1. To find out whether transparency processing is enabled for the current world, go to the World layout and look at the title of the World drawing window. If you have a color that needs to be opaque from the beginning but will be turned transparent during the experiment, set its value to 1 instead of NaN in order to ensure that transparency is enabled for the entire world.

ZOOMING IN AND OUT

- Zooming in and out on parts of the texture work the same way as on the World sketch. You can zoom either on the Texture sketch window or on the Texture drawing window. Both images of the texture will automatically change axis limits together.

SELECTING A SHAPE OR COLOR

- Selecting shapes and colors works the same way as selecting objects.

NAMING A SHAPE OR COLOR

- Naming shapes works the same way as naming objects using the *Rename shape* button.

DELETING A SHAPE OR COLOR

- Deleting shapes and colors works the same way as deleting objects using the *Delete shape* button.

SORTING SHAPES AND COLORS

- Sorting the names of shapes and colors works the same as sorting objects using the *Sort shapes* button.

CHANGING SHAPE OR COLOR PROPERTIES

- Changing shape and color properties works the same way as changing object properties.

CHANGING SHAPE OR COLOR LOCATIONS

- Changing shape or color locations works the same as changing object locations using the table in the Shape properties window.

ADDING AND DELETING SHAPE OR COLOR LOCATIONS

- Adding and deleting locations of shapes and colors works the same way as for objects, using the *Add shape location(s)* and *Delete shape location(s)* buttons.

ENTERING A SYMBOLIC LOCATION FOR A SHAPE OR COLOR

- You can enter a symbolic expression for the location of shapes the same way as you would for objects, using the *Symbolic shape location(s)* button.

COMPUTING A TEXTURE

- Once you sketch a texture and assign colors to all regions, the next step is to **triangulate** it. This will convert the sketch of the texture into a set of triangles that will be displayed by the ViRMEn engine. To triangulate, press the *Compute texture* button. Computing a texture may take some time depending on the complexity of your texture and the amount of triangulation refining.
- The computed triangulation will be displayed in the Texture drawing window. Temporarily switch to the *World layout* to see that the newly computed texture has been applied to your object.

DISPLAYING AND COLORING THE TRIANGULATION MESH

- By default, boundaries of the triangles (**mesh**) in the computed texture are shown. You can turn on/off the visibility of the mesh by pressing the *Show/hide triangulation* button.
- You can also change the color of the displayed mesh by pressing the *Triangulation mesh color* button and choosing a color.
- Note that the triangulation mesh is only displayed in the GUI for design purposes. The ViRMEn engine does not display the mesh during the experiment, and it is therefore invisible to the animal when the experiment is running.
- By default, mesh is visible and colored red. You can change these defaults by editing the default properties file.

REFINING TEXTURE TRIANGULATION

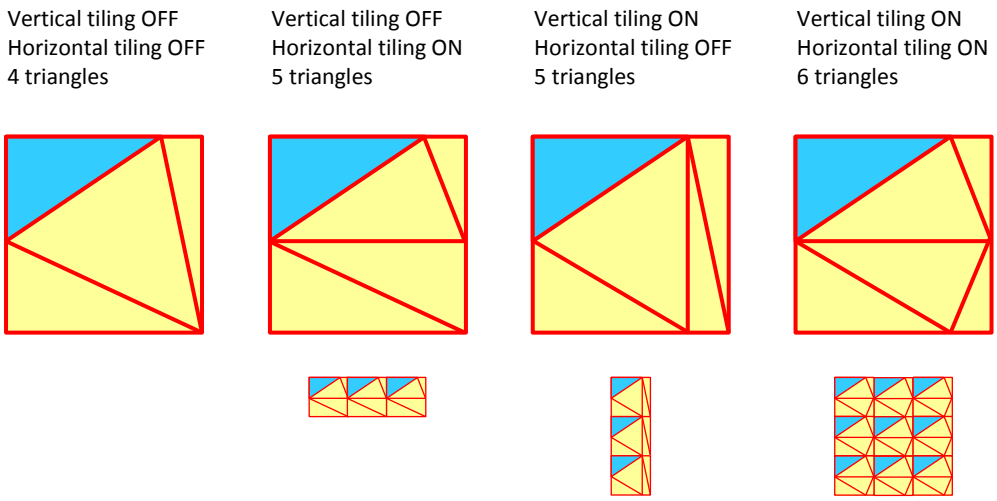
- The ViRMEn triangulation algorithm attempts to split the texture into the smallest possible number of triangles. This often creates triangles that are too large and cause objects to look too crude. For instance, if you applied a texture to a cylinder, you may have noticed that, from above, it looks more like a polygon than a circle. To solve this problem, you can **refine** triangulation – i.e., split a texture into a larger number of smaller triangles.
- Press the *Refine triangulation* button. A dialog box will ask for separate vertical and horizontal refining parameters. Each parameter indicates what maximum fraction of the texture a single triangle is allowed to occupy (along the x or the y axis). A smaller value thus represents finer triangulation. Determine which dimension of the texture you want to refine. For example, if you want a vertical

cylinder to look round and not have corners, the horizontal triangulation is more relevant. Reduce the triangulation parameter to, say, 0.2, then press the *Compute texture* button to recompute the triangulation.

- Of course, refining the triangulation increases the number of triangles. Unfortunately, this is the single most important determinant of the refresh rate of the ViRMEn engine. The number of triangles required by a texture is displayed in the title of the Texture drawing window. You can also see how many triangles your entire world has by going to the World layout and looking at the title of the World drawing window. If you plan to design a large or complex world, you may have to find a compromise between the fineness of triangulation and the refresh rate of the engine.
- By default, both vertical and horizontal refining parameters are set to 1 (least refined). You can change these defaults by editing the default properties file.

CONTROLING TEXTURE TILABILITY

- By default, ViRMEn assumes that you will be tiling the texture both vertically and horizontally on the surface of an object. It therefore ensures that every triangle vertex located on the top edge of the texture matches a vertex in the corresponding location on the bottom edge. Similarly, it matches locations of vertices located on the left and right edges of the texture. If you don't plan to tile the texture along one (or both) or the directions, you can turn **tilability** on and off using the *Vertical tiling on/off* button and the *Horizontal tiling on/off* button. This often reduces the number of triangles used by the texture. The following image shows the same texture triangulated with different combinations of vertical and horizontal tiling.
- Note that if you tile a texture on the surface of an object but neglect to turn tiling on, the object may contain visible "gaps" at the locations where tiles meet.
- By default, both vertical and horizontal tilability options are turned on. You can change these defaults by editing the default properties file.



Effects of turning vertical and horizontal tilability on or off on a texture. Note that the ViRMEn algorithm matches points along opposite edges of the texture to allow for seamless tiling.

IMPORTING A TEXTURE

- You can import a texture that you've previously created. To do so, press the *Import texture* button. A window will appear, giving you the list of all previously created experiment names.
- Select the experiment from which you wish to import a texture. Images of all available textures will appear. Click on the texture you wish to import.

LOADING A TEXTURE IMAGE FROM A FILE

- To load a texture from an image file, press the *Load image* button and select a file. ViRMEn will load an image and convert it into a ViRMEn texture. Press the *Compute texture* button to triangulate the texture, as usual.
- ViRMEn does represent an image as a matrix of pixels, but rather breaks an image into color regions and treats each region as a separate shape that is then triangulated. This is a computationally intensive operation. I do not recommend using texture image files larger than about 16x16 pixels or files that use more than a handful of distinct colors.
- Using an image file can substantially increase the number of triangles used to triangulate a texture. (For example, diagonal lines are represented very inefficiently in a pixelated image). Whenever possible, draw textures from scratch using the GUI.

USING THE SAME TEXTURE ON MULTIPLE OBJECTS

- When you import a texture, the list of experiment names includes the current experiment. Importing a texture from the current experiment will duplicate it. Use this to apply a texture of one object to another object in the same experiment.
- Whenever you change an object's texture, ViRMEn identifies other objects that had identical textures prior to the change. If such objects exist, a menu will pop up, allowing you to select any subset of these objects whose textures should also change. Use this to simultaneously change the textures of several objects.

TILING TEXTURE ON AN OBJECT

- At this point, switch to the World layout of the GUi to visualize the entire object you've applied the texture to. In the Object properties window, there are two textboxes for the amount of **vertical tiling** and **horizontal tiling**. These numbers indicate how many times the texture is repeated on the surface of the object. Change these numbers and observe the effect on your object.
- If you change the size of the object (e.g., by increasing the height or radius of the cylinder), the texture will stretch with it. For example, if vertical tiling is set at 5, each tile will take up 1/5 of the height of the cylinder, independently of the height. It is, however, possible to have the texture not scale with the object by using custom variables.
- By default, both vertical and horizontal tiling values are set to 5. You can change these defaults by editing the default properties file.

VIEWING AND EXPORTING

CHANGING THE WORLD BACKGROUND COLOR

- To change the background color of a world, activate the World drawing window and press the *World background color* button. The color you select will be used as background both in the GUI and during the actual experiment.
- By default, world background is set to black. You can change the default by editing the default properties file.

VIEWING AND ROTATING THE WORLD IN 3D

- You can interactively rotate the image of the world by pressing the *3D world view* button. An image of your world will take up the entire figure and the Matlab rotation tool will be activated.
- You can also change the 3D view of your world by selecting the World drawing window and using one of several buttons on the toolbar: *Isometric view*, *Top view*, *Front view*, and *Side view* buttons switch the view to a corresponding angle; *Rotate counterclockwise*, *Rotate clockwise*, *Rotate down*, and *Rotate up* buttons rotate the view by 45°.

EXPORTING IMAGES OF WORLDS AND TEXTURES

- You can export images to external Matlab figures. This is useful for creating figures and presentations. Press on the *Export* dropdown menu. You have the choice of exporting a 2D sketch of the world, the full 3D rendering of the world, or an image of the current object's texture.

CUSTOM VARIABLES

The properties and locations of all objects and shapes in ViRMEn don't have to be specified as numbers, but can be defined in terms of custom variables. Furthermore, ViRMEn will recognize any Matlab expression in these definitions. This is a very powerful tool that you can use to allow very fast and seamless changes to your worlds. The following examples show you how to use variables and suggest possible applications.

EXAMPLE: FIXING THE ASPECT RATIO OF AN OBJECT

- Add a Floor object to your world. Change the location to (0, 0). In the Object properties table, change the width to 100 and the height to 200.
- Now, suppose you want to change the size of the object you've created, but you want to keep its height:width ratio at 2:1. Rather than changing both the width and the height each time, we will define both dimensions in terms of a variable we will call floorWidth.
- In the Object properties table, for the value of the width, enter floorWidth. ViRMEn will recognize it as a new variable and prompt you for its value. Enter 100.
- In the Object properties table, for the value of the height, enter 2*floorWidth.
- Press the *Variables table* button. A table will pop up, containing the variable floorWidth and its value. Change the value and observe the change to the object. The object should scale without a change in aspect ratio.

EXAMPLE: LOCKING OBJECTS TO EACH ANOTHER

- Custom variables can be used to link multiple objects. After completing the previous example, add vertical cylinder to the world.
- Suppose you want the cylinder to always be located at the corner of the floor. In the Locations table, change the x location to floorWidth/2 and the y location to –floorWidth. The cylinder will be placed in the bottom-right corner of the floor.
- Open the variables table by pressing the *Variables table* button. Change the value of floorWidth and confirm that the cylinder is locked to the floor.

EXAMPLE: SCALE-FREE TILING OF A TEXTURE

- Add a floor object to your world and define both its width and height as 100. Now assign any texture to this object and make sure that both the vertical and the horizontal tiling are set to 5. The texture should be tiled 25 times in a square.
- Now change the floor width to 200. The texture remains tiled 25 times, but stretches horizontally. This may not be the behavior you want. You might want the individual texture tiles to remain square, but the total number of these squares to change.
- If order to achieve scale-free tiling, enter floorWidth for the width of the floor. ViRMEn will recognize floorWidth as a new variable and prompt you for its value. Enter 100.
- For horizontal tiling, enter floorWidth/20. The texture will be tiled 5 times horizontally.
- Press the *Variables table* button. A table will pop up, containing the variable floorWidth and its value. Change the value to 200. As the floor expands, ViRMEn should add additional texture tiles to it instead of stretching tiles.

INSERTING AND DELETING VARIABLES

- Press the *Experiment layout* button to switch to the Experiment layout. The Custom variables table contains the list of all variables you added to your world. Note that if you imported anything into your current experiment from other experiments, you might have imported some associated variables as well.
- To delete variables you no longer need, select the corresponding row(s) in the table and press the *Delete variable(s)* button. If you delete a variable that's still in use by the experiment, ViRMEn will prompt you for its value and restore it to the experiment.
- To add a new variable, press the *Add variable* button. You can enter variables that are not used by any object in your world, but are rather parameters you will later use in your code. This way, when you run an experiment, you can change relevant parameters using the table, without having to edit the Matlab code each time.

SORTING VARIABLES

- ViRMEn lists custom variables in the order in which they were created. To re-organize variable names into any order you wish, press the *Sort variables* button. A window will pop up, allowing you to move individual variable names up or down, or to sort variable names alphabetically.

RUNNING THE WORLD

At this point, you should have enough knowledge of the interface to design and draw a world. We will now learn how to run this world using the ViRMEn engine.

TRANSFORMATION AND MOVEMENT FUNCTIONS

- Go to the Experiment layout of the GUI. The Experiment properties window in this layout contains dropdown lists that allow you to select external functions to be used by your experiment.
- Select a movement function from one of the dropdown lists. A movement function is used by the ViRMEn engine to obtain velocity information. For initial testing, chose a movement function that allows you to control velocity using the computer mouse or keyboard. See movement functions for details.
- Select a transformation function from another dropdown list. The transformation function is specific to the geometry of the projection you're using for the virtual reality. See transformation functions for details.
- You can open the movement or transformation function in the Matlab editor by choosing appropriate options from the *Edit code* dropdown menu, or by pressing the *Edit* buttons next to the dropdown lists in the Experiment properties window.
- When you create a new experiment using the ViRMEn GUI, dropdown menus will be set to the default transformation and movement functions. You can change these defaults to determine which transformation and movement functions are automatically selected on your experimental setup.

ANTIALIASING

- The ViRMEn engine is capable of antialiasing rendered images of the world. To implement antialiasing, ViRMEn renders the world several times, each time jittering all coordinates by an amount smaller than 1 pixel. All of the rendered images are averaged to produce a single image that is displayed by the engine.
- The Experiment properties window allows you to enter the number of times you want the ViRMEn engine to re-render the world. The higher the number, the better the quality of antialiasing. However, antialiasing is very time consuming and requires a fast computer with a good graphics card. Start with no antialiasing (value 0) and successively try higher numbers – values of 4 or 8 usually work well.
- By default, antialiasing is set to 0. You can change the default by editing the default properties file.

SAVING AN EXPERIMENT

- To save an experiment, press the *Save experiment* button. You will be prompted for the name of a .mat file to save the experiment to. Place this .mat file in the "experiments" subfolder of ViRMEn. The .mat file contains a single variable exper, containing all information about the worlds, objects, textures, etc. in your experiment.
- When you save an experiment, ViRMEn also creates a blank template .m file that will contain all of the custom Matlab code that you write for your experiment. You can edit this file by choosing "Experiment code" from the *Edit code* dropdown menu.

OPENING OR CREATING A NEW EXPERIMENT

- To open a previously created experiment, press the *Open experiment* button.
- To erase the current experiment and start over with a blank experiment, press the *New experiment* button.
- When you open a .mat experiment file, ViRMEn attempts to find the .m file that contains all associated Matlab code. If this file cannot be found, ViRMEn creates it and populates it with code recovered from the codeText property. This provides a convenient way to share experiments with other people – you only need to share a single .mat file.

RUNNING AN EXPERIMENT

- To run your experiment, press the *Run* button. ViRMEn will start the engine.
- The virtual reality display will take up the full screen. Depending on your settings, part of the display might be blocked by the Windows taskbar. To prevent this, I suggest right-clicking on the taskbar, going to Properties, and selecting the "Auto-hide the taskbar" option.
- To stop the engine, press Esc or click the cancel button in the upper-right corner of the screen (the button auto-hides when you move the mouse cursor away). You can also stop the engine at a desired time using the vr.experimentEnded field in Matlab code you write.

ADDING PHYSICAL EDGES TO OBJECTS

- Create a world with at least one object and run the engine. Move around the world to get a feel for how the display updates. Now try running into one of the objects. You should be able to run through the object. This is because ViRMEn by default does not treat objects as physical entities. To create a physical boundary around the object, you will need to use edges.
- To add an edge to an object, go to the World layout and select one of the objects in the world. In the Object properties window, find a textbox labeled "Edge radius." By default, the edge radius value is NaN, indicating that the object does not have an edge. Change this value to 5. You will see that a dotted line appears around the object. This line indicates an edge that cannot be crossed by the animal. The ViRMEn engine performs continuous-time collision detection (CTCD). If an animal's displacement vector attempts to cross one of the edges, the vector is projected onto the edge, causing the animal to "skid" along the edge. (You can, however, control the skidding behavior with code).
- Play around with the edge radii. Then run the engine again and attempt to run through one of the objects.
- By default, object edges are turned off (i.e., radii are set to NaN). You can instead set a default radius value by editing the default properties file.

RUNTIME PROGRAMMING

Runtime programming refers to all custom code you write that will be executed while the experiment is running. This section walks you through code organization and some basic examples.

VIRMEN ENGINE PIPELINE

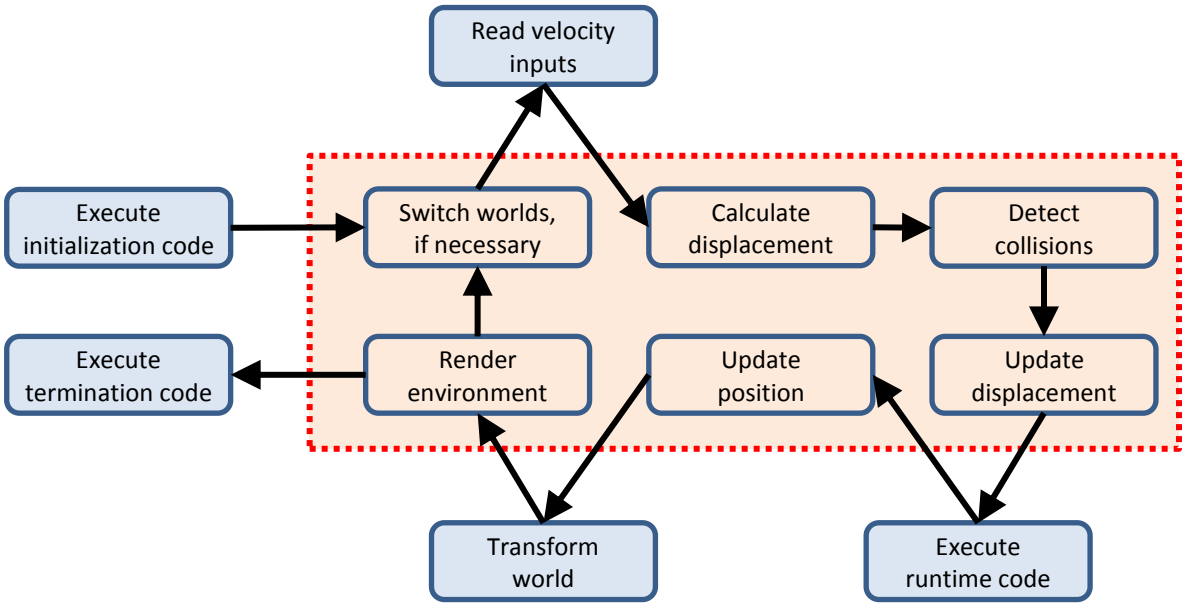


Diagram of the steps executed by the ViRMEn engine. Routines in the red dotted box are part of the ViRMEn engine itself. The five routines outside of this box are not part of the engine and can be customized by the user.

To begin writing custom code for ViRMEn, it is important to understand what exactly ViRMEn does and in which order. The diagram above illustrates the operation of the ViRMEn engine. Here is a brief explanation of each of the steps.

1. **Execute initialization code.** The initialization code is a custom function written by the user. It can contain routines like initialization of a DAQ, initialization of data log files, initialization of custom variables that will be used during the experiment, etc. It is run once before the engine starts.
2. **Switch world, if necessary.** If the world has been switched during the last engine iteration (by changing the value of vr.currentWorld), or if no world has been loaded yet, ViRMEn loads the world indicated by vr.currentWorld into memory.
3. **Read velocity inputs.** ViRMEn runs an external movement function, which is custom-written by the user. The movement function provides ViRMEn with velocity values. It can obtain those velocity values from any input device, such as a DAQ, a mouse, or a keyboard. The field vr.velocity is updated.
4. **Calculate displacement.** ViRMEn calculates displacement by multiplying velocity values by the amount of time elapsed since the last iteration. The field vr.dp is updated.
5. **Detect collisions.** ViRMEn runs continuous-time collision detection (CTCD) to determine if the animal is currently colliding with any of the physical edges in the world. The field vr.collision is updated.

6. **Update displacement.** If a collision between an animal and a physical edge is detected, ViRMEn corrects the displacement vector vr.dp by projecting it onto the physical edge. This causes the animal to "skid" along edges.
7. **Execute runtime code.** The runtime code is a custom function written by the user. This function should contain most of the routines related to the experiment, such as all the experiment logic, triggering of rewards, logging data to a file, writing outputs to DAQ, teleporting, real-time world manipulations, etc.
8. **Update position.** ViRMEn updates the animal's position by adding the displacement to it. The field vr.position is updated.
9. **Transform world.** ViRMEn transforms the 3D virtual world coordinates into 2D coordinates that will be displayed on the computer monitor or projector. The transformation is a function written by the user and is customized to the particular screen shape and projection geometry used.
10. **Render world.** ViRMEn renders the world using OpenGL graphics commands. Antialiasing is implemented during rendering.
11. **Execute termination code.** If the value of vr.experimentEnded is set to true, or if the user has pressed the Esc button or clicked the cancel button, the engine run is terminated. Termination code is a custom function written by the user. If can include routines like termination of a DAQ, closing of files, displaying summary information about the experiment, etc.

INTRODUCTION TO CODING VIRMEN EXPERIMENTS

For each experiment, you need to write three Matlab functions: initialization function, runtime function, and termination function. Initialization and termination functions are run once each – before the ViRMEn engine starts running and after the ViRMEn engine stops running. The runtime function runs once on each iteration of the engine. See pipeline for more details.

For convenience, all three functions are stored in a single .m file. This file is automatically generated whenever you save a new experiment with the ViRMEn GUI. A new blank template file has the following structure.

```
function code = sampleCode
% sampleCode   Code for the ViRMEn experiment sampleCode.
%   code = sampleCode   Returns handles to the functions that ViRMEn
%   executes during engine initialization, runtime and termination.

% Begin header code - DO NOT EDIT
code.initialization = @initializationCodeFun;
code.runtime = @runtimeCodeFun;
code.termination = @terminationCodeFun;
% End header code - DO NOT EDIT

% --- INITIALIZATION code: executes before the ViRMEn engine starts.
function vr = initializationCodeFun(vr)

% --- RUNTIME code: executes on every iteration of the ViRMEn engine.
function vr = runtimeCodeFun(vr)

% --- TERMINATION code: executes after the ViRMEn engine stops.
function vr = terminationCodeFun(vr)
```

All three functions use a Matlab structure variable called vr as both input and output. The values in this structure persist throughout an entire engine run. Therefore, vr is a convenient place to store any custom variables that need to be accessed and/or changed during an experiment (see example below).

You can change the default template file so that it differs from the one shown above. See customizing the template file for details.

EXAMPLE: KEEPING TRACK OF THE NUMBER OF REWARDS

A good way to keep track of the number of rewards is to define a variable numRewards as a field of the structure vr. The ViRMEn engine will keep the value stored in this variable from one iteration to the next, and will also make it available after the engine stops running. The following example defines the variable in the initialization code, updates it using the runtime code, and displays the value after the engine stops using the termination code

In the initialization code

```
vr.numRewards = 0; % define and initialize the variable numRewards
```

In the runtime code

```
if reward_condition % test if the reward condition has been satisfied
    vr.numRewards = vr.numRewards + 1;
end
```

In the termination code

```
% display the number of rewards received while the engine was running
disp(['The animal received ' num2str(vr.numRewards) ' rewards.']);
```

EXAMPLE: SHARING CODE BETWEEN EXPERIMENTS

Many routines, such as initializing a DAQ or defining certain variables, will be common to multiple experiments (or even multiple users on a given experimental setup). I recommend placing such common routines into external functions, and running these functions from within the initialization, runtime, or termination code. Typically, an external function you write will use the structure vr as both an input and an output. The following example assumes that multiple experiments use a variable numRewards and write data to a file called log.dat. The statements that initialize numRewards and open the file are therefore placed into a separate function. The statements that display the number of rewards and close the file are also placed into a separate function.

In the initialization code

```
vr = commonInitialization(vr); % call an external function
```

In the termination code

```
vr = commonTermination(vr); % call another external function
```

In a separate file

```
function vr = commonInitialization(vr)

% define and initialize a variable numRewards
vr.numRewards = 0;

% open a log file and store the file identifier in vr
vr.fid = fopen('log.dat');
```

In another separate file

```
function vr = commonTermination(vr)

% display the number of rewards received while the engine was running
disp(['The animal received ' num2str(vr.numRewards) ' rewards.']);
```

```
% close the log file
vr.fid = fclose(vr.fid);
```

PROGRAMMING THE RUNTIME "VR" STRUCTURE

In addition to custom variables, vr contains built-in fields that can be accessed by the user. These fields store real-time information about the engine, such as the animal's current position, velocity, etc. These fields can be changed by the user to trigger real-time changes – for example, changing vr.position will instantaneously teleport the animal through the world. The following is a list of all built-in fields or vr.

- **vr.exper**: a variable of the class virmenExperiment that contains all information about the experiment, worlds, etc. Changing this variable during the experiment does not do anything. However, I recommend saving this variable for future reference each time you run an experiment – that way you will have a record of all day-to-day changes you might have made to your experiment.
- **vr.code**: a structure containing function handles of the initialization, runtime, and termination functions.
- **vr.worlds**: a cell array of structures containing all information about the worlds used by the current experiment. Changing these structures will cause instantaneous changes to worlds. See the section on manipulating worlds for details.
- **vr.experimentEnded**: a Boolean indicating whether the experiment should be forced to end. Setting this to true will immediately stop the engine and run the termination function.
- **vr.currentWorld**: the index of the current world. This must have an integer value from 1 to the length of vr.worlds. Changing this value will instantaneously switch worlds.
- **vr.position**: an array of length 4 containing the current position in the form [x, y, z, viewAngle]. The units are ViRMEn space units for x, y, and z and radians for the viewAngle. Changing these values will instantaneously teleport or rotate the animal in the world.
- **vr.velocity**: an array of length 4 containing the current velocity in the form (d/dt)[x, y, z, viewAngle]. The units are ViRMEn space units/sec for the first three values and radians/sec for the fourth value. Changing these values does not do anything; to effect the animal's movement, change vr.dp instead.
- **vr.dt**: the amount of time in seconds elapsed since the previous iteration. Changing this value does not do anything.
- **vr.dp**: the change in position and view angle being implemented on the current iteration. Typically, vr.dp=vr.velocity*vr.dt. However, if the animal is currently in collision with an edge of an object, vr.dp is projected onto the edge, so that the animal "skids" along it. Changing the value of vr.dp will change the amount by which the animal moves and/or rotates on the current iteration.
- **vr.collision**: Boolean indicating whether the animal is currently in collision with an edge of an object.
- **vr.text**: a structure containing all textboxes displayed on the screen. Changing this structure can be used to introduce new textboxes or modify or delete existing ones.
- **vr.textClicked**: the index of the textbox that was clicked by the user during the previous iteration. If no textbox was clicked, the value is NaN.
- **vr.keyPressed**: the character of a key that was pressed by the user during the last iteration. If no key was pressed, the value is NaN.
- **vr.iterations**: the number of iterations performed by the engine during the current run.
- **vr.timeStarted**: Matlab time stamp indicating the time point at which the engine started running.

- **vr.window**: handle of the Windows Form containing the ViRMEn window. See help from Microsoft for programming this form.
- **vr.oglControl**: handle of the OpenGL control containing the ViRMEn display. See help from OpenTK for programming this control.

EXAMPLE: TELEPORTING

The animal can be teleported by changing vr.position within the runtime code. The following code will teleport the animal to the beginning of a linear track once he reaches the end of the track. We assume that the track is oriented along the y dimension of the environment and has length 200. While teleporting the animal, we also set the displacement vector vr.dp to 0; this is necessary to prevent any additional movement during teleportation. (Because vr.dp was calculated before teleportation, it may not be valid at the new location.)

In the runtime code

```
if vr.position(2) > 200 % test if the animal is at the end of the track (y > 200)
    vr.position(2) = 0; % set the animal's y position to 0
    vr.dp(:) = 0; % prevent any additional movement during teleportation
end
```

EXAMPLE: SWITCHING WORLDS

Worlds can be switched by changing the value of vr.currentWorld. The following code will switch the animal between two linear tracks. On each trial, the animal runs on one of two linear tracks (worlds 1 and 2). As soon as he reaches the end of the track (length 200 along the y direction), he is teleported to the beginning of a randomly chosen track.

In the runtime code

```
if vr.position(2) > 200 % test if the animal is at the end of the track (y > 200)
    newWorldIndx = ceil(rand*2); % choose a random world index from 1 to 2
    vr.currentWorld = newWorldIndx; % set the current world
    vr.position(2) = 0; % set the animal's y position to 0
end
```

EXAMPLE: CONTROLING SKIDDING ALONG WALLS

When a collision with an edge of an object is detected, the animal's displacement vector is projected onto the edge. In other words, the animal "skids" along the edge with no friction. It is possible to control the skidding behavior by manipulating the value of vr.dp.

In the initialization code

```
vr.friction = 0.3; % define friction that will reduce velocity by 70% during collisions
```

In the runtime code

```
if vr.collision % test if the animal is currently in collision
    % reduce the x and y components of displacement
    vr.dp(1:2) = vr.dp(1:2) * vr.friction;
end
```

In the above code, vr.friction = 1 will produce the default frictionless behavior. A value of vr.friction = 0 will completely block movement whenever collision happens.

EXAMPLE: USING CLICKABLE TEXTBOXES

Some parts of the computer monitor are typically invisible to the animal because they do not project onto the virtual reality screen. These regions can be used to display useful information to the experimenter, such as the amount of time elapsed or the number of rewards received. The structure vr.text can be used to create and update textboxes. This structure contains fields string, position, size and color. The string can contain capital letters A-Z, digits 0-9, spaces, as well as the following characters: - + * / = % . , ( ) [ ]. Position and size are both in units of screen height/2, such as the top of the screen is at 1 and the bottom is at -1. The center of the monitor is at (0, 0).

The following example displays the time elapsed since the beginning of the engine run.

In the initialization code

```
% Define a textbox and set its position, size and color
vr.text(1).position = [-1.2 1]; % upper-left corner of the screen
vr.text(1).size = 0.03;
vr.text(1).color = [1 1 0]; % yellow
```

In the runtime code

```
% On every iteration, update the string to display the time elapsed
vr.text(1).string = ['TIME ' datestr(now-vr.startTime,'MM.SS')];
```

Textboxes can also be programmed to respond to mouse clicks. That way, they act like buttons that can allow the experimenter to trigger manual rewards, teleport the animal, flash stimuli, etc. The value of vr.textClicked indicates which textbox has been clicked during the last iteration of the engine. If no textbox has been clicked, the value is NaN. For example, adding these lines to the previous example will generate a sound each time the textbox is clicked.

In the runtime code

```
if vr.textClicked == 1 % check if textbox #1 has been clicked
    beep
end
```

Note that you can also make ViRMEn respond to user input via keyboard using the vr.keyPressed field.

EXAMPLE: ACCESSING VALUES OF CUSTOM VARIABLES

The ViRMEn GUI allows the user to define custom variables that describe various features of the virtual world. Sometimes, it may be necessary to access the values of these variables during the experiment itself. This can be done by accessing vr.exper.variables. The following example assumes that a variable trackLength has been used to define the length of a linear track. Whenever the animal reaches the end of the linear track, he is teleported to the beginning. Using the custom variable guarantees that the code will work properly whenever trackLength is changed using the GUI in order to shorten or lengthen the track.

In the initialization code

```
% evaluate the expression for trackLength
vr.trackLength = eval(vr.exper.variables.trackLength);
```

In the runtime code

```
if vr.position(2) > vr.trackLength % check if the animal's y position exceeds trackLength
    vr.position(2) = 0; % teleport the animal to the beginning of the track
end
```

TRANSFORMATION FUNCTIONS

A transformation function determines how 3D coordinates within the virtual world are transformed into 2D coordinates that the projector displays on the screen. A transformation function is specific to each experimental setup and is determined by the shape of the screen (flat, toroidal, conical, etc.) and the geometry of the projection (projector angle, mirror shape, etc.). Transformations functions recognized by ViRMEn must be .m or mex files placed in the "transformations" subfolder of ViRMEn. The following is the sequence of steps used by the ViRMEn engine to render a world on the screen. This should help you understand how to derive a transformation function for your experimental setup.

1. Whenever a world is loaded by the ViRMEn engine, it is *triangulated* – converted in to a set of triangles defined by their vertices and colors. Consider one of the vertices, (x, y, z). Our goal is to understand where this vertex should appear on the computer monitor, and consequently in the projected image.
2. On every iteration of the engine, the animal's location in the virtual world is denoted by $(x_a, y_a, z_a, \theta_a)$. Here θ_a is the view angle. The ViRMEn engine first subtracts the animal's location from the vertex location to produce the *translated* (subscript *T*) version of the coordinates (x_T, y_T, z_T), where $x_T = x - x_a$, $y_T = y - y_a$ and $z_T = z - z_a$. These are the coordinates of the vertex in the reference frame in which the animal is at the origin.
3. The vertex location is then rotated around the z axis in the opposite direction of the animal's view angle to produce the *translated and rotated* (subscript *T&R*) version of the coordinates $(x_{T\&R}, y_{T\&R}, z_{T\&R})$, where $[x_{T\&R} \quad y_{T\&R}]^T = R(-\theta_a)[x_T \quad y_T]^T$, where $R(-\theta_a)$ is the rotation matrix of angle $-\theta_a$. The resulting coordinates are the coordinates of the vertex in the reference frame in which the animal is at the origin and facing in the direction of 0 degrees.
4. Steps 2 and 3 illustrate how ViRMEn handles animal's movements: instead of modifying the viewpoint on every iteration, it moves and rotates the entire world, while keeping the viewpoint stationary. This simplifies the 3D->2D transformation, because it allows us to assume that the animal is always at the origin facing in the 0-degree direction. The transformation function then converts the 3D coordinate $(x_{T\&R}, y_{T\&R}, z_{T\&R})$ into a 2D monitor/projector coordinate (x_P, y_P). In Matlab, the transformation function takes a 3xN matrix as input, with rows containing $x_{T\&R}$, $y_{T\&R}$ and $z_{T\&R}$. As output, it returns a 3xN matrix where the first two rows contain x_P and y_P. The third row of the output matrix contains 0's or 1's indicating whether the location is visible. This is necessary for making invisible those locations that are outside the projected image. The monitor/projector coordinate system is defined to be (0, 0) at the center of the monitor; the monitor is defined to be 2 units high, such that the top of the monitor is at $y_P = 1$ and the bottom of the monitor is at $y_P = -1$. (The limits of x_P depend on the aspect ratio of the monitor.
5. In the final step, ViRMEn sorts all triangles according to their distance from the animal. Triangles that are farther from the animal are rendered first, while triangles close to the animal are rendered last. This way, if two or more triangles overlap in the projected image, the triangle closest to the animal occludes those located farther away.

Transformation functions are often computationally intensive functions because they operate on large matrices of coordinates. Furthermore, the transformation function in use is called by the ViRMEn engine on every iteration and can therefore substantially slow down the engine refresh rate. I recommend coding transformation functions as mex files rather than Matlab routines.

As an example, see the Appendix for the derivation of transformation functions for radially symmetric screens.

MOVEMENT FUNCTIONS

On each iteration, the ViRMEn engine calls a *movement function*, which returns values indicating the current velocity of the animal in the world. These velocity values are then used to update the animal's position and view angle in the world. Typically, a movement function will read data from analog input channels and appropriately filter and/or scale them. Alternatively, a movement function can calculate velocity using inputs from a mouse or keyboard.

Movement functions recognized by ViRMEn must be .m of mex files placed in the "movements" subfolder of ViRMEn. A movement function takes the structure vr as input. As output, it returns a vector of 4 values in the form (d/dt)[x, y, z, viewAngle]. The units the first three values are ViRMEn space units per second; the units for the fourth value are radians per second.

EXAMPLE: MOVING FORWARD WITH CONSTANT VELOCITY

The following example is a movement function that moves the animal forward in the world (along the y direction) with a constant velocity. The value of velocity can be defined in the initialization code without having to change the movement function each time.

In the initialization code

```
vr.forwardVelocity = 20;
```

In the movement function

```
function velocity = moveForward(vr)

velocity = [0 vr.forwardVelocity 0 0];
```

REAL-TIME WORLD MANIPULATIONS

You can program your runtime code to manipulate worlds in in real time by modifying values in the vr.worlds field on the vr structure. This section will show you how to work with vr.worlds by demonstrating several simple examples of world manipulations.

The field vr.worlds Is a cell array containing information about all worlds in the current experiment. To make real-time changes to the current world, we will make changes to vr.worlds{vr.currentWorld}

EXAMPLE: MAKING THE WORLD INVISIBLE

Suppose we want to "turn off" the world and place the animal in complete darkness. The ViRMEn engine represents the entire worlds as a set of triangles, which are stored in the "surface" structure:

```
>> vr.worlds{vr.currentWorld}.surface

ans =

         vertices: [3x10793 double]
    triangulation: [3x19862 int32]
          visible: [1x19862 logical]
           colors: [4x10793 double]
```

This example world contains 10793 triangle vertices, whose coordinates are store in the "vertices" field (the 3 rows are x, y, and z coordinates). These vertices form 19862 triangles (many of vertices are shaped by multiple triangles.

The matrix in the "triangulation" field indicates indices of the triplets of vertices that are linked into triangles. Each triangle has a visibility Boolean assigned to it, indicated by the "visible" field. To make the entire world invisible, set all of the visibility Booleans to false:

In the runtime code

```
vr.worlds{vr.currentWorld}.surface.visible(:) = false; % make all triangles invisible
```

EXAMPLE: MOVING AN OBJECT

As mentioned above, the field vr.worlds{vr.currentWorld}.surface.vertices contains the coordinates of all triangles in the world. By changing these coordinates, we can instantaneously move objects in the world.

Suppose we have an object in the current world called targetCylinder, and we want to move this object 5 units along the y axis. To do so, we should change the 2nd row of vr.worlds{vr.currentWorld}.surface.vertices. However, the world contains 10793 vertices, and only some of them belong to targetCylinder. Thus, we first need to find out which of the vertices we should change. To obtain this information, we will use the "objects.vertices" subfield:

```
>> vr.worlds{vr.currentWorld}.objects.vertices

ans =

           1        7357
        7358       10169
       10170       10467
       10468       10793
```

These values indicate that there are 4 objects in the current world. The first object occupies vertices 1 through 7357, the second object occupies vertices 7358 through 10169, and so on. One of these four objects is our targetCylinder. To find out which one, we will use the "objects.indices" subfield:

```
>> vr.worlds{vr.currentWorld}.objects.indices

ans =

               roomWall: 1
    objectCircularFloor: 2
         targetCylinder: 3
                cueCard: 4
```

The object named targetCylinder is the 3rd one, meaning that it occupies vertices 10170 through 10467. (This is why it's important to assign meaningful names to all your objects). If we change columns 10170 through 10467 of vr.worlds{vr.currentWorld}.surface.vertices, we will instantaneously move targetCylinder without affecting any other objects. Putting all this together (note that the code which doesn't need to get repeated on each iteration is placed in the initialization code):

In the initialization code

```
% determine the index of targetCylinder
indx = vr.worlds{vr.currentWorld}.objects.indices.targetCylinder;

% determine the indices of the first and last vertex of targetCylinder
vertexFirstLast = vr.worlds{vr.currentWorld}.objects.vertices(indx,:);

% create an array of all vertex indices belonging to targetCylinder and store it in vr
vr.cylinderIndx = vertexFirstLast(1):vertexFirstLast(2);
```

In the runtime code

```
% determine the current y coordinates of targetCylinder vertices
```

```
        y = vr.worlds{vr.currentWorld}.surface.vertices(2, vr.cylinderIndx);

        % increase the y coordinates by 5 units
        vr.worlds{vr.currentWorld}.surface.vertices(2, vr.cylinderIndx) = y + 5;
```

EXAMPLE: CHANGING OBJECT TRANSPARENCY

To change the transparency or color of an object, we will modify the "color" subfield of vr.worlds{vr.currentWorld}.surface:

```
>> vr.worlds{vr.currentWorld}.surface

ans =

         vertices: [3x10793 double]
    triangulation: [3x19862 int32]
          visible: [1x19862 logical]
           colors: [4x10793 double]
```

The 4 rows of the "colors" matrix contain the RGB values and the Alpha (transparency) value. If the world does not contain any transparent objects, the matrix will only have 3 rows containing RGB values (see changing transparency for details). Each column of the "colors" matrix indicates the color and transparency of the corresponding vertex. (Triangles in OpenGL are colored according to their last vertex; for instance, a triangle composed of vertices [6 20 83] will be colored with the color assigned to vertex 83).

The following code sets the transparency of targetCylinder to 70%.

In the initialization code

```
        % determine the index of targetCylinder
        indx = vr.worlds{vr.currentWorld}.objects.indices.targetCylinder;

        % determine the indices of the first and last vertex of targetCylinder
        vertexFirstLast = vr.worlds{vr.currentWorld}.objects.indices(indx,:);

        % create an array of all vertex indices belonging to targetCylinder and store it in vr
        vr.cylinderIndx = vertexFirstLast(1):vertexFirstLast(2);
```

In the runtime code

```
        % set the transparency of targetCylinder to 0.7
        vr.worlds{vr.currentWorld}.surface.colors(4,vr.cylinderIndx) = 0.7;
```

INPUT-OUTPUT METHODS

The ViRMEn engine does not include any built-in features for reading and outputting data. These features must be written in Matlab and included in your code. This section provides examples for how to integrate simple input/output methods into custom ViRMEn code.

EXAMPLE: INTERFACING WITH A DAQ

To interface with a DAQ, you need the Matlab Data Acquisition Toolbox installed. Refer to Matlab documentation for how to install and register your DAQ hardware in Matlab.

The easiest way to deal with a DAQ is to define all input and output channels in the initialization code of your custom ViRMEn code. Handles of all input and output objects should be stored in the vr structure. That way, all functions run by ViRMEn during the experiment (e.g., the runtime function and the movement function) will have access to these objects. The following is sample code illustrating how to define 2 analog input objects for use in ViRMEn.

In the initialization code

```
% reset DAQ in case it's still in use by a previous Matlab program
daqreset;

% connect to the DAQ card dev1; store the input object handle in vr for use by ViRMEn
vr.ai = analoginput('nidaq','dev1');

% start analog input channels 0 and 1
addchannel(vr.ai,0:1);

% define the sampling rate to 1kHz and set the duration to be unlimited
set(vr.ai,'samplerate',1000,'samplespertrigger',inf)

% set the buffering window to be 8 ms long - shorter than a single ViRMEn refresh cycle
set(vr.ai,'bufferingconfig',[8 100]);

% define a temporary log file to be deleted at the end of the experiment
set(vr.ai,'loggingmode','Disk');
vr.tempfile = [tempname '.log'];
set(vr.ai,'logfilename',vr.tempfile);

% start acquisition from the analog input object
start(vr.ai);
```

In the termination function

```
% stop the analog input object
stop(vr.ai);

% delete the temporary log file
delete(vr.tempfile);
```

I recommend using the Matlab commands peekdata for reading data from an analog input object and putsample for writing data to an analog output object. These are both non-blocking functions that will not slow down the execution of the ViRMEn engine. Refer to the Matlab help on these functions. Data can be read or written in the runtime function or your custom code. In addition, you can read data from a DAQ within your movement function in order to obtain velocity information from an input device (such as an optical mouse).

EXAMPLE: LOGGING TO A FILE

Writing to a binary file in Matlab is fast an can be implemented within the runtime function or your Matlab code. Use this to log behavioral data – or other relevant information – to a file. The following example shows how to log the animal's time-stamped position and velocity and how to reconstruct these data after you're done with the experiment.

In the initialization code

```
% open or create binary file for writing and store its file ID in vr
vr.fid = fopen('virmenLog.dat','w');
```

In the runtime code

```
% obtain the current timestamp
timestamp = now;

% write timestamp and the x & y components of position and velocity to a file
% using floating-point precision
```

```
        fwrite(vr.fid, [timestamp vr.position(1:2) vr.velocity(1:2)],'double');
```

In the termination code

```
        % close the file
        fclose(vr.fid);
```

External code used to log and plot behavioral data

```
        % open the binary file
        fid = fopen('virmenLog.data');

        % read all data from the file into a 5-row matrix
        data = fread(fid,[5 inf],'double');

        % close the file
        fclose(fid);

        % plot the 2D position information
        plot(data(2,:),data(3,:));
```

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming allows you to manipulate and display virtual worlds programmatically with Matlab scripts without using the ViRMEn GUI. This is a powerful tool that allows you to code experiments that would be too time-consuming to create with the GUI and gives you more flexibility for making complex changes. For this section, I suggest some knowledge of object-oriented programming and the Matlab class structure.

Object-oriented commands allow you to create experiments and worlds entirely from scratch. Alternatively, they let you make changes to an experiment you created with the ViRMEn GUI. To do so, load the .mat file stored in the "experiments" subfolder of ViRMEn. This file contains a single variable **exper**, which is a variable of class virmenExperiment (described below).

VIRMEN CLASSES

ViRMEn object-oriented programming includes 5 variable classes:

1. virmenExperiment – ViRMEn experiment
2. virmenWorld – environment (world)
3. virmenObject – 3D object (such as a vertical cylinder)
4. virmenTexture – texture of an object
5. virmenShape – 2D shape (such as rectangle) used in textures

Classes virmenObject and virmenShape are superclasses containing individual object and shape types. For example, the virmenVerticalCylinder class is a subclass of virmenObject, and shapeRectangle is a subclass of virmenShape. All recognized objects classes are defined in the "objects" subfolder of ViRMEn. All recognized shape classes are defined in the "shapes" subfolder of ViRMEn.

CREATING VIRMEN CLASS VARIABLES

You can create **experiments**, **worlds**, and **textures** by simply calling the corresponding ViRMEn class name. For example, the following code creates a ViRMEn experiment:

```
>> ex = virmenExperiment

ex =

  virmenExperiment handle

  Properties:
              antialiasing: 0
          movementFunction: @undefined
    transformationFunction: @undefined
           experimentCode: @undefined
                 codeText: {}
                   worlds: {[1x1 virmenWorld]}
                     name: 'virmenExperiment'
                   parent: {}
                    items: [1x1 struct]
                 symbolic: [1x1 struct]
                variables: [1x1 struct]

  Methods, Events, Superclasses
```

To create **objects** or **shapes**, call the particular object or shape class type. For example, the following code creates a vertical cylinder object:

```
>> c = objectVerticalCylinder

c =

  objectVerticalCylinder handle

  Properties:
            radius: 5
            bottom: 0
        startAngle: 0
         stopAngle: 360
               top: 40
                 x: [0x1 double]
                 y: [0x1 double]
            tiling: [5 5]
        edgeRadius: NaN
           texture: [1x1 virmenTexture]
     iconLocations: [2x2 double]
              name: 'objectVerticalCylinder'
            parent: {}
             items: [1x1 struct]
          symbolic: [1x1 struct]
         variables: [1x1 struct]

  Methods, Events, Superclasses
```

To create a **color** to add to a texture, use the built-in shapeColor class:

```
>> c = shapeColor

c =

  shapeColor handle

  Properties:
                 R: []
                 G: []
                 B: []
             Alpha: NaN
                 x: [0x1 double]
                 y: [0x1 double]
     iconLocations: [2x2 double]
              name: 'shapeColor'
            parent: {}
          symbolic: [1x1 struct]
         variables: [1x1 struct]

  Methods, Events, Superclasses
```

VIRMEN CLASS PROPERTIES

All ViRMEn classes have properties assigned to them. These properties are easy to set. For example, the following code creates a world and changes its background color to white:

```
>> w = virmenWorld;
>> w.backgroundColor = [1 1 1]

w =

  virmenWorld handle
```

```
  Properties:
    backgroundColor: [1 1 1]
      startLocation: [0 0 0 0]
            objects: {}
               name: 'virmenWorld'
             parent: {}
              items: [1x1 struct]
           symbolic: [1x1 struct]
          variables: [1x1 struct]

  Methods, Events, Superclasses
```

Object and shape classes have type-dependent properties. For example, a variable of type objectVerticalCylinder has a property named radius. To obtain a list of all properties for a given class, use the Matlab properties command:

```
>> properties(objectVerticalCylinder)

Properties for class objectVerticalCylinder:

    radius
    bottom
    startAngle
    stopAngle
    top
    x
    y
    tiling
    edgeRadius
    texture
    iconLocations
    name
    parent
    items
    symbolic
    variables
```

The properties can also be specified using custom variables. Just set the property value to a string containing a mathematical expression. For example, to create a vertical cylinder and set its radius to a variable 'r' (see symbolic property for more details):

```
>> c = objectVerticalCylinder;
>> c.radius = 'r';
```

The following sections are a reference of the properties of all ViRMEn.

VIRMENEXPERIMENT PROPERTIES

- **antialiasing**: level of antialiasing used for rendering the experiment's worlds
- **movementFunction**: Matlab function handle of the experiment's movement function
- **transformationFunction**: Matlab function handle of the experiment's transformation function
- **experimentCode**: Matlab function handle of the file containing the experiment's initialization, runtime, and transformation functions
- **codeText**: cell array of strings containing all the code from the experiment's .m file. Note that if you change the .m file, the codeText property does not automatically update. To update the codeText property, use the updateCodeText method.
- **worlds**: cell array of virmenWorld variables containing all of the experiment's worlds

VIRMENWORLD PROPERTIES

- **backgroundColor**: background color of the world (1x3 array)
- **startLocation**: the animal's starting location in the world (1x4 array in the form [x, y, z, viewAngle])
- objects
- **objects**: cell array of virmenObject variables containing all of the world's objects

VIRMENOBJECT PROPERTIES

Properties of ViRMEn objects depend of the object class type. For instance, a variable of class objectVerticalCylinder has a property called radius. For reference on these properties, refer to the particular object type. In addition, there are properties of all ViRMEn objects, independent of the object type, listed below.

- **x**: x component of the object's locations (Nx1 array for N locations or a scalar if all x coordinates are the same)
- **y**: y component of the object's locations (Nx1 array for N locations or a scalar if all y coordinates are the same)

  For example, to create 2 vertical cylinders and placed at (0, 0) and (100, 200):

```
>> c = objectVerticalCylinder;
>> c.x = [0; 100];
>> c.y = [0; 200];
```

- **tiling**: amount of texture tiling on the surface of the object (1x2 array in the form [vertical horizontal])
- **edgeRadius**: radius of the physical edge around the object (NaN for no edge)
- **texture**: virmenTexture variable containing the object's texture
- **iconLocations**: matrix containing xy locations of the object for display in the icon representing the object in the *Add object* dropdown menu of the ViRMEn GUI (Nx2 matrix)

VIRMENTEXTURE PROPERTIES

- **width**: the width of the texture. Typically set to 1 and not changed.
- **height**: the height of the texture. Typically set to 1 and not changed.
- **tilable**: Booleans indicating whether the texture is tilable (1x2 array in the form [vertical horizontal])
- **refining**: the texture's triangulation refining parameters (1x2 array in the form [vertical horizontal])
- **shapes**: cell array of virmenShape variables containing all of the texture's shapes
- **triangles**: structure containing information about all of the triangles making up the texture's triangulation. Note that the triangles property does not automatically update if you make changes to the texture. To update the triangles property, use the compute method. The triangles structure contains the following fields

  vertices: coordinates of all triangle vertices (Nx2 matrix of x and y values)
  triangulation: indices of vertex triplets that are linked together to create triangles (Mx3 array matrix of triplet). A single vertex can be shared by multiple triangles.
  cdata: color and transparency values of all triangle vertices (Nx4 matrix of RGB and Alpha values). A triangle is assigned the color and transparency of its last (3rd) listed vertex.

VIRMENSHAPE PROPERTIES

Properties of ViRMEn shapes depend of the shape class type. For instance, a variable of class shapeRegularPolygon has a property called radius. For reference on these properties, refer to the particular shape type. In addition, there are properties of all ViRMEn shapes, independent of the shape type, listed below.

- **x**: x component of the shape's locations (Nx1 array for N locations)
- **y**: y component of the shape's locations (Nx1 array for N locations)
- **iconLocations**: matrix containing xy locations of the shape for display in the icon representing the shape in the *Add shape* dropdown menu of the ViRMEn GUI (Nx2 matrix)

PROPERTIES COMMON TO ALL VIRMEN CLASSES

The following are properties carried by variables of any ViRMEn class type. For instance, any ViRMEn item – be it a virmenExperiment, virmenWorld, etc – has a property called "name".

- **name**: the item's name. When a ViRMEn variable is created, it is automatically given a name according to the variable class (e.g., "virmenExperiment", "virmenWorld", etc.)
- **parent**: handle of the item one step above the variable in the hierarchy (virmenExperiment → virmenWorld → virmenObject → virmenTexture → virmenShape). For example, the parent of a virmenWorld is a virmenExperiment variable; the parent of a virmenTexture is a virmenObject variable. The parent of a virmenExperiment variable is an empty array.
- **symbolic**: structure containing all of the item's properties in symbolic form. This is useful for looking up expressions of the objects properties that were specified using custom variables.

  For example, suppose we want to create a vertical cylinder and assign it a radius indicated by a custom variable r. To do so, you can simply set the object's radius property to the string 'r'. Once you do that, you will be prompted for the value of the new variable r. Enter 10 for the value.

```
>> c = objectVerticalCylinder;
>> c.radius = 'r';
```

  Now, when you prompt for the radius property of the cylinder, the value 10 is returned

```
>> c.radius

ans =

    10
```

  However, when you prompt for the symbolic property, the string 'r' is returned

```
>> c.symbolic.radius

ans =

r
```

  To change the radius to a different symbolic expression, change the radius property (not the symbolic property):

```
>> c.radius = '2*r';
>> c.radius

ans =

    20
```

- **variables**: a structure containing all custom variable names and values.

  For instance, after running the above example, you will find the variable r in this structure:

```
>> c.variables

ans =

    r: '10'
```

You can set the variable to a new value, which must be a string containing an expression. This will automatically update all properties that depend on the variable. For instance, in the above example, changing the variable r will automatically change the radius of the cylinder (which has been set to 2*r):

```
>> c.variables.r = '100';
>> c.radius

ans =

   200
```

Note that all variables are shared by members of the same hierarchy. For instance, each of the objects inside a world will have the same variables structure as the world itself.

VIRMEN CLASS METHODS

VIRMENEXPERIMENT METHODS

- **addWorld**

  addWorld(e, w) adds the virmenWorld w to the virmenExperiment e. To delete a world from the experiment, simply remove it from the e.worlds array.

- **updateCodeText**

  updateCodeText(e) updates the codeText property of the virmenExperiment e by loading the code from the .m file

- **run**

  err = run(e) runs the the virmenExperiment e in the ViRMEn engine. Variable err contains any errors that were produced during the engine log.

VIRMENWORLD METHODS

- **draw3D**

  h = draw3D(w) creates a 3D rendering of the virmenWorld w in a Matlab figure. Output h is the handle of the 3D surface.

- **draw2D**

  [ho he ha] = draw2D(w) creates a 2D top-view sketch of the virmenWorld w in a Matlab figure. The outputs ho contains handles of all objects sketched, he contains handles of all edges around objects, and ha contains the handle of the square-and-arrow symbol representing the animal's starting location in the world.

- **addObject**

  addObject(w, obj) adds the virmenObject obj to the virmenWorld w. To delete an object from the world, simply remove it from the o.objects array.

- **triangulate**

tri = triangulate(w) creates a structure tri containing the triangulation of the world. This structure is the same as those stored in vr.worlds.

VIRMENOBJECT METHODS

- **locations**

  loc = locations(obj) returns an Nx2 matrix of the locations of the virmenObject obj.
- **draw3D**

  h = draw2D(obj) creates a 3D rendering of the virmenObject obj in a Matlab figure. Output h is the handle of the 3D surface.
- **draw2D**

  [ho he] = draw2D(obj) creates a 2D top-view sketch of the virmenObject obj in a Matlab figure. The output ho contains the handle of the object sketch, and he contains the handle of the edge around the object.
- **setTexture**

  setTexture(obj, t) sets virmenTexture t as the texture of virmenObject obj.

VIRMENTEXTURE METHODS

- **draw**

  h = draw(t) draws a triangulated image of virmenTexture t. The output h is a handle of the created surface object.
- **sketch**

  h = sketch(t) creates a un-triangulated sketch of virmenTexture t. The output h contains the handles of all shapes contained in the texture.
- **compute**

  compute(t) computes virmenTexture t. Running this method updates the triangles property of the texture.
- **addShape**

  addShape(t, s) adds the virmenShape s to the virmenTexture t. To delete a shape from the texture, simply remove it from the t.shape array. Note that adding or deleting a shape will not automatically update the triangulation of the texture. To update triangulation, run the compute method (see above).
- **loadImage**

  loadImage(t, img, col) converts an indexed image img (2D matrix) with colormap col (Nx3 matrix) to a ViRMEn texture t. The texture is represented as a set of colored regions, but is initially not triangulated. To triangulate, run the compute method (see above).

VIRMENSHAPE METHODS

- **locations**

  loc = locations(s) returns an Nx2 matrix of the locations of the virmenShape s.

METHODS COMMON TO ALL VIRMEN CLASSES

The following are methods allowed for any ViRMEn class type. For instance, any ViRMEn item – be it a virmenExperiment, virmenWorld, etc – has a method called "fullName".

- **ancestor**

  Returns the handle of the top-level item in the ViRMEn class hierarchy (virmenExperiment → virmenWorld → virmenObject → virmenTexture → virmenShape). Usually, the ancestor of any ViRMEn item will be the virmenExperiment that this item is contained in.

- **children**

  Returns the handles of all items one level lower in the ViRMEn class hierarchy (virmenExperiment → virmenWorld → virmenObject → virmenTexture → virmenShape). For example, virmenWorld items are the children of a virmenExperiment, whereas a virmenTexture item is the child of a virmenObject.

- **descendants**

  Returns the handles of all items lower in the ViRMEn class hierarchy (virmenExperiment → virmenWorld → virmenObject → virmenTexture → virmenShape). For example, the descendants of a virmenWorld will include all virmenObject handles in that world, the virmenTexture handles assigned to those objects, and the virmenShape handles that make up those textures.

- **fullName**

  Returns a string containing the full name of an item. This is normally the same as the name property of the item, except if there exist multiple items with the same name. For example, if there are two objects with the name "myCylinder", the method fullName will return strings 'myCylinder(1)' and 'myCylinder(2)' for those objects.

- **copyVariable**: copy a ViRMEn class variable to a different variable:

  All ViRMEn classes are Matlab handles. This means that copying them is somewhat different from copying other variable types, such as arrays. For example, suppose you create a vertical cylinder object as a variable named c1 and set its radius to 10:

```
>> c1 = objectVerticalCylinder;
>> c1.radius = 10;
```

  Now suppose you create a copy of variable c1 and name it c2. Because c1 and c2 are handles, they point to the same object, and any change you make to one of them will now affect the other:

```
>> c2 = c1;
>> c1.radius = 50;
>> c2.radius

ans =

    50
```

  If instead of having two handles that point to the same object, you want to duplicate the object itself, use the **copyVirmenObject** command:

```
>> c1 = objectVerticalCylinder;
>> c1.radius = 10;
>> c2 = copyVirmenObject(c1);
```

  Now, c1 and c2 refer to different objects, and changing one will have no effect on the other:

```
>> c1.radius = 50;
>> c1.radius
```

```
ans =

    50

>> c2.radius

ans =

    10
```

CUSTOMIZING VIRMEN

EDITING GUI LAYOUTS

The ViRMEn GUI has three built-in layouts: Experiment, World and Texture. Each of these layouts displays a set of windows at particular positions within the GUI figure. You can customize the positions where these windows are displayed to suit your preferences. In addition, you can create your own custom layouts that display any subset of windows you wish. Information about all layouts recognized by the ViRMEn GUI is stored in the file **defaultLayouts.txt** in the "defaults" subfolder of ViRMEn. This file contains a tab-delimited table and is easiest to edit with a program like MicrosoftExcel. To return this file to the "factory" default state, delete it and restart ViRMEn. ViRMEn will automatically create a new defaultLayouts.m file matching the "factory" default file.

MODIFYING THE EXPERIMENT, WORLD AND TEXTURE LAYOUTS

To edit built-in layouts (Experiment, World and Texture), we will change values in the row entitled "Default" of the table in defaultLayouts.txt.

The Experiment layout includes three windows: Custom variables, Experiment properties, and Worlds menu. These are hard-wired into the Experiment layout – you cannot remove any of these three windows or add any window other than these three. (If you want more flexibility, see creating a custom layout). To change the position of one of these windows within the Experiment layout, find the name of the window among column headers of the table. Underneath, you will find sub-headers titled x, y, w, and h. These specify the bottom-left corner of the window (x, y), the width w, and the height h in normalized units.

Similarly, the World layout contains three windows: Object properties, World sketch, and World drawing. Changing the coordinates of any of these windows will affect the World layout. The Texture layout also contains three windows: Shape properties, Texture sketch, and Texture drawing.

CREATING A CUSTOM LAYOUT

You can create your own layout other than the three built-in ones (Experiment, World and Texture). To do so, add a new row to the table in defaultLayouts.txt. In the first column of the new row, type in the title of your layout. To include a window in your layout, fill in the four values under the header of the desired window name: x, y, w, and h, specifying the bottom-left corner of the window (x, y), the width w, and the height h of the window in normalized units. For windows that you don't want included in your layout, fill all cells with dashes (-).

For example, suppose you wanted to create a layout named "myLayout" that included only the Experiment properties window, and that you wanted this window to fill up the entire figure. The table would look like this (the values you would add to the existing table are emphasized in red):

-	variablesTable	-	-	-	experimentProperties	-	-	-	worldsMenu	-	-	-	objectProperties	-	-	-	...
Name	x	y	w	h	x	y	w	h	X	y	w	h	x	y	w	H	...
Default	0	0.5	1	0.5	0	0	0.25	0.5	0.25	0	0.75	0.5	0	0	0.25	1	...
myLayout	-	-	-	-	0	0	1	1	-	-	-	-	-	-	-	-	...

OPENING A CUSTOM LAYOUT

After creating a custom layout, restart the ViRMEn GUI. Your new layout will be available in the *Layout* dropdown menu.

EDITING THE DEFAULT TEMPLATE CODE

When you save a newly created experiment, ViRMEn automatically creates a "template" .m file for your Matlab code containing the initialization, runtime, and termination functions. You can change the default template file – for instance, to include code that you always want to have in all of your experiments. The default template file is called **defaultVirmenCode.m** and located in the "defaults" subfolder of ViRMEn. Changing this file will affect every new experiment you create. To return this file to the "factory" default state, delete it and restart ViRMEn. ViRMEn will automatically create a new defaultVirmenCode.m file matching the "factory" default file.

CHANGING DEFAULT MOVEMENT AND TRANSFORMATION FUNCTIONS

The movement function and the transformation function are typically specific to a particular experimental setup and never change. You may therefore want to set the default movement and transformation functions so that the ViRMEn GUI assigns them automatically to new experiments you create. Default functions are listed in the **defaultFunctions.txt** file located in the "defaults" subfolder of ViRMEn. This is a tab-delimited text file. Simply change names of functions to those you want to use. To return functions to "factory" defaults, delete the defaultFunctions.txt file and restart ViRMEn. ViRMEn will automatically create a new defaultFunctions.txt file with "factory" defaults.

CHANGING DEFAULT PROPERTY VALUES

You can customize certain default properties to change the behavior of the ViRMEn GUI. The following table lists these properties. Default property values are listed in the **defaultProperties.txt** file located in the "defaults" subfolder of ViRMEn. This file is a tab-delimited text file best edited with a program like Microsoft Excel. To return all default values to "factory" defaults, delete the defaultProperties.txt file and restart ViRMEn. ViRMEn will automatically create a new defaultProperties.txt file with all "factory" defaults.

Property	**Explanation**	**"Factory" default value**
antialiasing	amount of antialiasing	0 (no antialiasing)
worldXLim	limits of the x-axis on the world sketch	[-100 100]
worldYLim	limits of the y-axis on the world sketch	[-100 100]
worldBackgroundColor	world background color	[0 0 0] (black)
startLocation	starting location and view angle of the animal, in the form [x, y, z, viewAngle]	[0 0 0 0]
tiling	amount of texture tiling on the surface of an	[5 5]

	object, in the form [vertical horizontal]	
edgeRadius	radius of a physical boundary (edge) around an object	NaN (no edge)
triangulationRefining	triangulation refining parameter, in the form [vertical horizontal]	[1 1]
showTriangulation	Boolean indicating whether the triangulation mesh of a texture should be visible	1
triangulationColor	color of the triangulation mesh	[1 0 0] (red)
textureTilable	Booleans indicating whether a texture is tilable, in the form [vertical horizontal]	[1 1] (tilable in both dimensions)

CREATING NEW OBJECT TYPES

To create a new object type, you need to write a Matlab class definition describing that object. Refer to Matlab help for all class programming information. The class must be a subclass of the virmenObject class. In order to be recognized by ViRMEn, the class definition file should be placed in the "objects" subfolder of ViRMEn.

The class can have any custom properties you wish your object to have, as well as the default values of these properties. The properties will be recognized by the ViRMEn GUI, and the user will be able to edit them. All properties should have the SetObservable meta-property set to true.

In addition, the class definition must contain 5 required methods, described below.

1. **creation method**. This method must have the same name as the class itself. It will be executed each time the object of your class is created. You might want to set iconLocations in this method. (The iconLocations property will determine how your class is displayed as an icon in the *Add objects* dropdown menu of the ViRMEn GUI.)
2. **getPoints**. This method must query the user for object locations and set the x and y properties of the class.
3. **coords2D**. This method must return the 2D coordinates of the object that will be drawn when the object is sketched in the top view. The form is [x y] = coords2D(obj); x and y should be Nx1 arrays containing the x and y coordinates of the line segments.
4. **coords3D**. This method should return a structure containing the triangulation of your object. The form is st = coords3D(obj). The structure st must have the following fields:
   vertices: coordinates of all triangle vertices (Nx2 matrix of x and y values)
   triangulation: indices of vertex triplets that are linked together to create triangles (Mx3 array matrix of triplet). A single vertex can be shared by multiple triangles.
   cdata: color and transparency values of all triangle vertices (Nx4 matrix of RGB and Alpha values). A triangle is assigned the color and transparency of its last (3rd) listed vertex.
5. **edges**. This method must return the coordinates of all physical edges associated with the object. The format edge = edges(obj) should return an Nx4 matrix, where every row is in the form [x1 y1 x2 y2]. If x1=x2 and y1=y2, ViRMEn will draw a circular edge of specified radius around (x1, y1). If these value are not equal, ViRMEn will draw an edge at the specified radius around a line segment connecting (x1, y1) and (x2, y2).

To see some examples, looks at some shape classes already available in the “objects” subfolder of ViRMEn. Once you create a new object class, restart ViRMEn. Your class should appear in the *Add object* dropdown menu.

CREATING NEW SHAPE TYPES

To create a new shape type, you need to write a Matlab class definition describing that shape. Refer to Matlab help for all class programming information. The class must be a subclass of the virmenShape class. In order to be recognized by ViRMEn, the class definition file should be placed in the “shapes” subfolder of ViRMEn.

The class can have any custom properties you wish your shape to have, as well as the default values these properties. These properties will be recognized by the ViRMEn GUI, and the user will be allowed to edit them. All properties should have the SetObservable meta-property set to true.

In addition, the class definition must contain 3 required methods, described below

1. **creation method**. This method must have the same name as the class itself. It will be executed each time the shape of your class is created. You might want to set iconLocations in this method. (The iconLocations property will determine how your class is displayed as an icon in the *Add shapes* dropdown menu of the ViRMEn GUI.)
2. **getPoints**. This method must query the user for shape locations and set the x and y properties of the class.
3. **coords2D**. This method must return the actual 2D coordinates of the line segments that make up the shape. The form for calling this method on shape s is [x y] = coords2D(s); x and y should be Nx1 arrays containing the x and y coordinates of the line segments.

To see some examples, looks at some shape classes already available in the “shapes” subfolder of ViRMEn. Once you create a new shape class, restart ViRMEn. Your shape should appear in the *Add object* dropdown menu.

APPENDIX

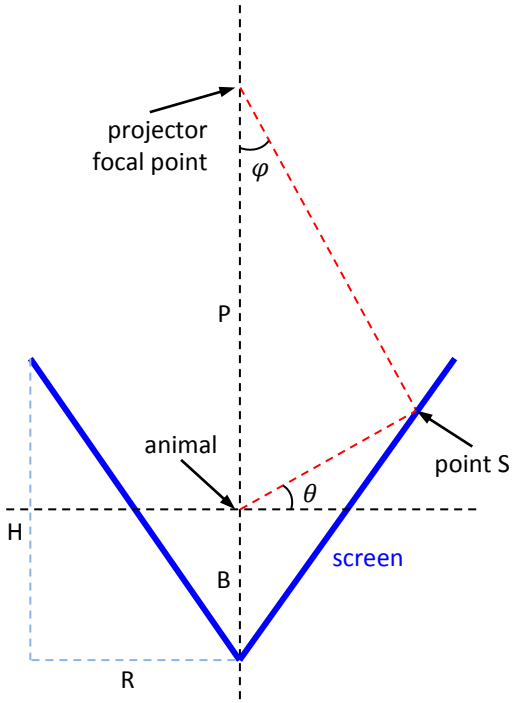
DERIVATION OF A TRANSFORMATION FUNCTION

We will derive a transformation function for the screen in the shape of an inverted cone. The procedure we will use is applicable for any other radially symmetric screen, such as a cylinder or a torus.

Note that ViRMEn is compatible with screens that are not radially symmetric, but you will have to derive a transformation function for each screen. For some particularly complex screens, it may be easiest to approximate a transformation function with empirical measurements (i.e., create a list of monitor locations and corresponding virtual reality screen locations, then fit relationship between them with some reasonably simple function).

At first, we define some constants that describe the shape of the screen and the projection geometry:

B vertical distance from the animal to the apex of the cone. (Note that the bottom of the cone can be truncated, but we will nonetheless derive the equation as if the entire cone were intact.)
H height of the screen
R radius of the screen at the top
P distance along the optical axis from the animal to the focal point of the projector
φ_{max} maximum angle from the optical axis that the projector can project in every direction



1. Suppose the projector projects a point at some angle φ from the optical axis. This point will be projected at a location *S* on the screen. To the animal, point *S* appears to be at some elevation angle θ above the horizon. Our first goal is to determine the relationship between angle φ and angle θ. Suppose point *S* is located at some radial distance r_P from the animal. This means that its vertical distance z_s from the animal is

$$z_s = r_S \tan\theta$$

The shape of the screen can be described by the linear equation

$$z = (H/R)r - B$$

Substituting the values of r_S and z_s,

$$r_S \tan\theta = (H/R)r_S - B$$

Solving this equation for r_S,

$$r_S = \frac{B}{H/R - \tan\theta}$$

From the projector geometry,

$$\tan\varphi = \frac{r_S}{P - r_S \tan\theta}$$

Substituting in the values of r_S from the previous equation,

$$\tan\varphi = \frac{1}{\frac{HP}{BR} + \left(1 - \frac{P}{B}\right)\tan\theta}$$

2. Now, we will determine how the angle φ is related to the location of a point on the monitor screen. ViRMEn normalizes monitor coordinates such that the shorter (vertical) monitor dimension extends from -1 to 1. This means that points at a radial distance of 1 from the center of the monitor display at a maximum projection angle of φ_{max}. Suppose that the projection angle of φ corresponds to a radial distance of r_M on the monitor.
   From trigonometry,

$$r_M = \tan\varphi / \tan\varphi_{max}$$

   Substituting these values into the previous equation

$$r_M = \frac{1}{\alpha + \beta\tan\theta}$$

   where

$$\alpha = \frac{HP}{BR}\tan\varphi_{max}$$

$$\beta = (1 - P/B)\tan\varphi_{max}$$

   Values of α and β are constants that can be calculated from the dimensions of a setup and the specifications of a projector – for instance, for my rat virtual reality setup, the values are $\alpha = 2.035$ and $\beta = -0.9899$.

3. We've now derived at an expression for the relationship between r_M and θ. This relationship can easily be transformed into a relationship between 3D world coordinates and 2D monitor coordinates. Suppose there is a point (x, y, z) is the 3D world. The tangent of the elevation angle can be expressed as

$$\tan\theta = z/r$$

   where $r = \sqrt{x^2 + y^2}$ (remember that the animal is defined to be always at the origin). Thus

$$r_M = r/(\alpha r + \beta z)$$

   This indicates that a radius r in the 3D world is scaled by $1/(\alpha r + \beta z)$ to produce radius r_M on the computer monitor. Because the projection is radially symmetric, the same **scaling factor** must apply to x and y components of the coordinates

$$x_M = x/\left(\alpha\sqrt{x^2 + y^2} + \beta z\right)$$

$$y_M = y/\left(\alpha\sqrt{x^2 + y^2} + \beta z\right)$$

   Where (x_M, y_M) are the coordinates of the point on the monitor screen. These equations describe the relationship between a point (x, y, z) in the world and a point (x_M, y_M) on the monitor; this is exactly what a transformation function requires.
   Putting this into Matlab code,

```
function coords2D = transformConical(coords3D)

% define constants that describe setup geometry
alpha = 2.0349;
beta = -0.98988;

% create an output matrix of the same size as the input
% first two rows are x and y
% the third row indicates whether the location should be visible
coords2D = zeros(size(coords3D));

% by default, make all points visible
coords2D(3,:) = true;

% calculate radius
r = sqrt(coords3D(1,:).^2 + coords3D(2,:).^2);

% calculate a scaling factor
s = 1./(alpha*r + beta*coords3D(3,:));

% if a point is outside of the screen, set the scaling factor such that the
% point is plotted at the edge of the screen, and make the point invisible
% (if all 3 vertices of a triangle are invisible, it is not plotted)
f = find(s < 0 | r.*s > 1);
s(f) = 1./r(f);
coords2D(3,f) = false;

% calculate x and y components using the scaling factor
coords2D(1,:) = s.*coords3D(1,:);
coords2D(2,:) = s.*coords3D(2,:);
```

EXTENSION TO ALL RADIALLY SYMMETRIC SCREENS: TOROIDAL EXAMPLE

For any other radially symmetric screen (e.g., cylinder or torus), it is necessary to calculate a different relationship between r_M and θ (see steps 1 and 2 in the above derivation). This does not have to be derived, but can be calculated empirically by plotting circles of varying radii r_M on the monitor and measuring at what elevation angles θ they appear on the virtual reality screen.

For example, for a mouse toroidal screen, this relationship was measured to be approximately linear

$$r_M = 0.4625\theta + 0.4929$$

(See Forrest Colman's Ph.D. dissertation).

Multiplying the right side of this equation by r/r, we again obtain version of the equation where the radius r is scaled to produce r_M

$$r_M = r[(0.4625\theta + 0.4929)/r]$$

Here, $r = \sqrt{x^2 + y^2}$ and $\theta = z/r$. The derivation of (x_M, y_M) then follows the same procedure as for the conical screen:

$$x_M = x[(0.4625\theta + 0.4929)/r]$$

$$y_M = y[(0.4625\theta + 0.4929)/r]$$

which are expressions of x_M and y_M in terms of x, y, and z required by transformation function.

The same process can be followed for any other radially symmetric screen.