Addition of Extended Gesture Controls and a Virtual Keyboard to the CDP Platform

Interdisciplinary Project

Documentation

Felix Lampe

Technische Universität München

March 2017

Contents

1 Introduction

The main feature of the Collaborative Design Platform (CDP) is the interactive table, which is mainly interacted with by means of touch gestures. With more and more features being introduced to the table, controlling all the features via menu entries becomes increasingly complicated, which leads away from an intuitive and tangible user interface. Therefore it has been decided that the gesture controls need to be extended with additional gestures and functions.

Apart from enhanced gesture controls, the addition of a digital keyboard was planned. However, because the on-screen keyboard provided by Windows has been proven to be easy to integrate during another Interdisciplinary Project (IDP), this feature was dropped in favor of an approach to make gestures dynamically configurable.

2 New Gestures

Four new gestures have been implemented. The first two, namely the Arc and Cross gestures have been chosen because of the need for gestures that can directly trigger certain functions on the table's main view ("TableView"). The Three Finger Tap and Four Finger Tap have been selected because they are easy to implement, they don't require the finger detection to be very precise, and they are intuitive to perform as a user.

As none of those gestures is provided by Qt, they were implemented as subclasses of QGesture, just like the already existing Double Tap gesture, which was implemented as part of an earlier IDP [1]. The implementations of all the custom gestures currently used by the CDP can be found inside the GestureRecognizer project.

2.1 Arc Gesture

The Arc gesture was implemented to provide a way of triggering a height scan of a physical building using the kinect mounted above the table. Adding a new gesture for that function had already been proposed in [1]. To perform the Arc gesture, the user has to draw a half circle using a single finger.

Inspired by the findings of [2] on user-defined gestures, the initial plan was to have a circle gesture which would be drawn around a building that should be scanned. However, this leads to an uncomfortable user experience when buildings are higher than the length of the users' fingers, because in that case one would have to use one's finger and forearm to draw the circle, which

could prove quite difficult for shorter users. A half circle does not have this restriction, because the finger does not have to move around the part of the building that is facing away from the user. This variation still allows precise recognition of the position the user was trying to indicate, because the center of the implied full circle is equal to the center of the half circle which is actually drawn.

The gesture recognition process for the Arc gesture works as follows:

1. The total angle sum is set to zero.
2. The positions of the points contained in the line are collected.
3. For every three consecutive points, the angle between the line segments connecting the first and second and the second and third points is calculated and added to the total angle sum.
4. When the finger is lifted, the gesture is accepted if
   (a) the angle sum is roughly 180° (a half circle) *and*
   (b) the distance from each point to the center between the first and last points is roughly the same.

2.2 Cross Gesture

Another gesture was needed for removing virtual buildings from view, a function which was not yet implemented. The two gesture ideas that seemed to be the most accepted among the CDP developers for such a function were again in line with [2]. Those gestures, namely a wiggly line and a Cross, can be seen in the sketches in Figure 1.

Figure 1: Sketches for gestures that could trigger a remove action.

The Cross has been selected for implementation because its shape is much better defined than that of the other gesture. For example, it would be very difficult to allow varying numbers and angles of turns in the wiggly line, while the Cross always consists of two lines that intersect at a right angle.

The gesture recognition process for the Cross gesture works as follows:

1. Two empty lists of points are created for the two lines.
2. While a finger moves across the table, its positions are added to the first list.
3. When the finger is lifted, the first line is marked as finished and points are added to the second list once a new touch happens.
4. When the finger is lifted the second time, the lines defined by the two point lists are interpolated using linear interpolation.
5. The intersection point and the angle of the interpolated lines are calculated.
6. The gesture is accepted if the intersection angle is roughly 90°.
7. Finally, the gesture's hotspot is set to the intersection point, and its radius is set to the maximum distance from the lines' ends to the intersection.

The interpolated lines are represented internally as a slope and a y-intersect in the table coordinate system.

2.3 Multi-Tap Gestures

Two additional gestures have been selected and implemented because of their simplicity, which makes not only the recognition process easy, but also memorizing them as a user. The Three Finger Tap and Four Finger Tap gestures are performed by simultaneously tapping with three or four fingers of one hand.

Their recognition processes are basically the same, and very simple:

1. The number of touch points is checked. If it is not three (four), the gesture is ignored.
2. The average position and orientation of the finger positions is calculated and set as the gesture's center and orientation, respectively.
3. The pairwise distance of the three (four) points are calculated. If one of them is beyond a certain threshold, the points are interpreted as not being part of the same gesture, and the gesture is ignored.
4. Otherwise, it is accepted.

These gestures have been added without a feature in mind that they should trigger. In fact, they have been added with the idea in mind, that gestures and functions might not need to be statically bound to each other, and could be configurable instead. This approach is detailed in section 4.

3 Building Hiding

Hiding virtual buildings shown on the table was implemented as a new function that can be triggered by gestures as described in sections 2.2 and 4.3. To allow hiding and restoring buildings, the class CDP_Document has been expanded to contain a list *virtualBuildingsOriginal* of all virtual buildings, shown and hidden, as a backup.

Hiding a building triggers the function *hideBuildings* of that class, which removes the provided buildings from the list of buildings that is exposed to other classes. The buildings are however kept in the backup list, which can be restored via the *restoreVirtualBuildings* function, replacing the contents of the public list.

The buildings that should be hidden are selected based on the distance from their center to that of the triggering gesture. The building class did already have a position attribute, which was however only used for translation and rendering when rotating the map, which is disabled as of writing this. The calculation of this position did not reliably happen in the existing code. Updating that calculation, as well as the usage in the translation, without breaking currently disabled functionality would have been too much effort, so it was decided to introduce a new attribute *center* instead, which holds the coordinates of the center of a building's vertices. Its value is calculated at object creation, so it can be used reliably later.

4 Configurable Gesture Controls

4.1 Static Gesture Handling

The existing gesture controls of the CDP consisted of predefined Qt gestures and a custom Double Tap gesture. Each of those was statically bound to one certain function that it would trigger when performed on a view. For example, double tapping when the main table view is active would always open up the pie menu.

This approach is very limiting, because users are forced to get used to gestures that they might not find intuitive, and there is no way for them to change that. Because of that, ways to dynamically define or configure gesture interactions were explored.

4.2 User-Defined Gestures

A gesture recognition system that learns gestures by example has already been proposed in the Future Work section of the previous IDP on touch

controls [1]. Such an interface would allow for very intuitive controls, great flexibility and extensibility, because users could create any gesture they like and new gestures could be easily added.

However, realizing such a concept is quite difficult, since many different aspects have to be thought of. For example, gestures that consist of multiple, unconnected lines, like the Cross gesture, are hard to recognize using only a shape comparison, as it is not clear whether the user is done drawing the gesture when they lift their finger the first time. That temporal aspect makes using Hidden Markov Models for recognition difficult, which would otherwise probably be the easiest way to implement a learning system, as was demonstrated in [3]. Implementations based on Neural Networks ([4], for example) work similarly, but have the same drawback.

Other solutions try to automate writing of the gesture recognition code instead of having the system actually learn on its own. Gesture Coder [5] for example generates Java code for a set of examples of a new gesture. While it does not allow to learn gestures at runtime, it still reduces the amount of work it takes to implement new gestures. However, integrating Java code with CDP's .NET codebase would be cumbersome, and it seems that Gesture Coder is not openly available anymore.

For these reasons, the idea of user-defined gestures has been abandoned for now, and another approach towards more dynamic gesture controls has been taken.

4.3 Configurable Gesture Mapping

Making the mapping of gestures to functions that can be triggered by them dynamically configurable adds a great amount of flexibility to the CDP too, and is much easier to achieve than the options mentioned before. A modifiable mapping allows users to bind the functions they use most to gestures they find easy and intuitive to perform. It also enables developers to change the function triggered by a gesture, even without changing the view to another with different functions. This could be valuable for applications like the PowerWall, which could be controlled using one gesture or menu entry to switch to a PowerWall control mode, during which gestures trigger actions on the PowerWall instead of on the table.

4.3.1 Classes and Structure

To implement the configurable gesture mapping, a number of new classes have been added to the GestureManager project. The most important one is GestureManager, which holds the mapping of gestures and functions in the

form of a stack of map objects. Each map binds values of a GestureID enum to values of a FunctionID enum, thus representing a configuration. The map on top of the stack is the currently active one that defines what functions are triggered if a gesture is performed. Whenever the configuration should be changed, a new maping can be activated by pushing a new map object onto the stack, or popping off the last one using the GestureManager's instance methods.

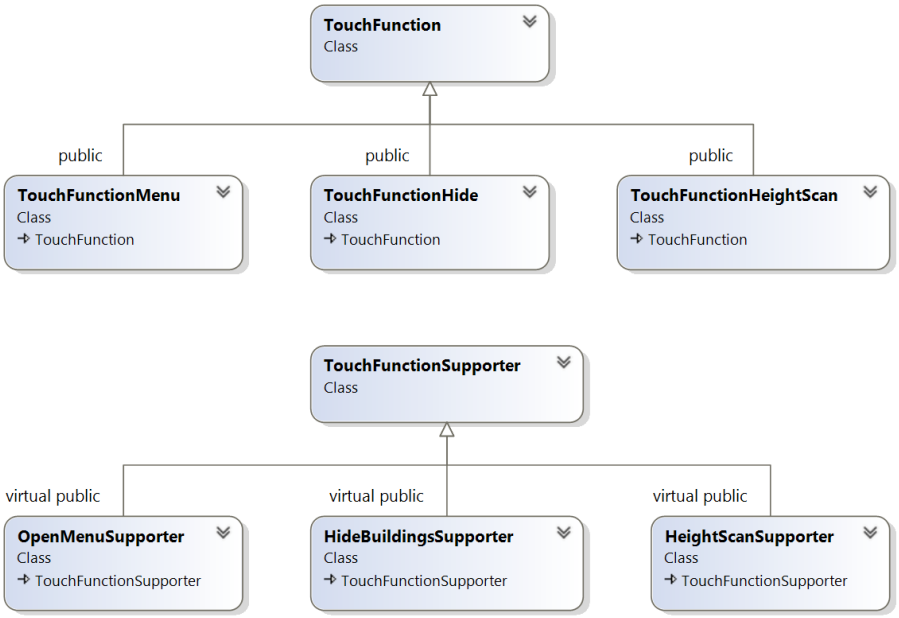

Figure 2: Class diagram showing part of the added class hierarchies

As of writing this, the dynamic binding is only used by the TableView class. All other views still handle gestures using their own dedicated code. The TableView however creates a GestureManager instance in its constructor, to which it delegates all gesture events. To make this possible, the TableView class has been made a subclass of three new interfaces that all derive from the new TouchFunctionSupporter interface. Those interfaces ensure that views that want to use a GestureManager provide the necessary methods. In the case of TableView, those functions are opening the menu, hiding virtual buildings and scanning building height.

For each of the TouchFunctionSupporter sub-interfaces, there is also a corresponding subclass of TouchFunction. Those classes retrieve values needed for the execution of the specific function from the performed gesture, and call the respective callback method in the view that the GestureManager is associated with. The view then decides how to interpret the values like position, orientation etc., and performs the actual function, updating its internal

data. This way, the logic that deals with specific gestures and functions is separated from the code that handles the mapping and configuration.

4.3.2 Configuration File

A hard-coded default mapping is active at startup of the CDP to ensure the views can always be controlled. The default behaviour can be changed using a configuration file that defines different mappings. The configuration file is read by the GestureManager when it is instantiated. That allows the user to change their preferences without having to recompile the framework with different hard-coded defaults.

The configuration file resides in *config/gestureConfig.json* (relative to the project's root folder). It is formatted in JSON to make it easy to read, edit and parse. It basically consists of a dictionary, or map, that links one gesture to one function. To change the defaults, replace the string value of a gesture or a function with another. The possible values are listed in the file as comments and should be updated when new gestures or touch functions are added.

```
{
    // Map gestures to functions.
    // Key must be a gesture, value a function.
    // Names are case-sensitive!
    //
    // Currently available gestures:
    // Arc, Cross, DoubleTap, ThreeFingerTap, FourFingerTap
    //
    // Currently available functions:
    // HeightScan, HideBuildings, Menu
    "mapping":
    {
        "Arc": "HeightScan",
        "Cross": "HideBuildings",
        "DoubleTap": "Menu",
        "ThreeFingerTap": "Menu",
        "FourFingerTap": "HideBuildings"
    }
}
```

Figure 3: Sample configuration file

Parsing the configuration file is relatively simple because of the JSON formatting. It is done in *loadConfigFromFile* in GestureManager.cpp, which basically looks for a JSON value named "mapping", iterates over its content, translates the gesture and function names to enum IDs and places them in

a map. This process is independent of the known gestures and touch functions, which means that the parser does not have to be changed when new gestures or functions are added, since it only relies on the enum entries and name maps.

4.3.3 Adding Views

To make another view use the GestureManager, the following steps need to be performed:

1. An attribute of type GestureManager has to be added to the view. It must be initialized in the view's constructor's initialization list, providing a reference to the view as the argument.
2. The view must be changed so that it inherits from all of the specific TouchFunctionSupporters, depending on which functions it should provide. The methods declared in the supporter interfaces have to be implemented as callback functions that use the gesture properties delivered as arguments in a meaningful way. See the method *scanHeight* in the TableView class for example.
3. In the view's *gestureEvent* method, the event handling must be delegated to the GestureManager attribute via its *triggerFunction* method, with the event's gesture as the parameter.

4.3.4 Adding Gestures

To add a new gesture that can be handled by GestureManager, the following steps are necessary:

1. Ensure that the gesture is a subclass of QGesture, like the existing custom made gestures included in the GestureRecognizer project.
2. Add a new entry in the GestureID enum in GestureManager.hpp and extend the method *createGestureNames* in the same file accordingly by mapping the ID to a name which will be used in the configuration file.
3. Change GestureManager's *triggerFunction* to select the GestureID you just added if it matches the handled gesture's type.
4. If any of the views that use GestureManager should react to the gesture, make sure they delegate the gesture event correctly for gestures of the new type.
5. Optional: edit the method *getDefaultMapping* in GestureManager.cpp to assign the new gesture a default action.

4.3.5 Adding Functions

To add a new function that should be triggered by GestureManager, follow these steps:

1. Add a new entry in the FunctionID enum in GestureManager.hpp and extend the method *createFunctionNames* in the same file accordingly by mapping the ID to a name which will be used in the configuration file.

2. Create subclasses of TouchFunction and TouchFunctionSupporter that represent the function and the interface for supporting views.

3. In TouchFunctionSupporter.hpp, define a method that represents the touch function that should be triggered. The method's parameter list must include all information necessary to perform the function, for example the orientation of the gesture.

4. Implement the execute method of the new TouchFunction subclass so that it gathers the gesture's properties that are relevant for the function and hands them over to the TouchFunctionSupporter.

5. If any of the views that use GestureManager should support the function, make them subclasses of the newly created TouchFunctionSupporter type, and implement the callback method in a way that makes sense for that view. Again, see TableView.cpp for examples.

6. Optional: edit the method *getDefaultMapping* in GestureManager.cpp to assign the new function to a default gesture.

5 Future Work

Regarding gestures, there are still multiple things left to improve. When drawing a line on the table with a finger, the scanned trace is sometimes interrupted, which creates new touch begin events instead of updating the existing one. This makes gestures relying on long lines hard to recognize, and should be prevented by improving the calibration and finger detection code.

Gestures could be made even more configurable by making constants used for recognition, like timeouts, angles and lengths, configurable via the configuration file.

The code structure and quality could be improved by splitting big files like CustomQGestureRecognizer in multiple parts, one for each contained class.

In GestureRecognizer, the *triggerFunction* method could be improved if the detection of the gesture subtype could be done in a general fashion, eliminating the need to try and cast the gesture object to each subtype until the cast is successful. In Java, this could be done via the Reflection API, but C++ does not provide this feature, and no other suitable solution could be found in this IDP.

Apart from the gesture related code, the building classes could be improved greatly regarding extensibility and readability. Some building properties are currently stored in a map from string to string that basically contains JSON key-value pairs. The keys however are not defined in one globally accessible spot, so that entries can have whatever key the programmer decides to use. If later the value should be used in another location, the developer has to search the entire project for uses of the property map, looking for hard-coded string keys. That even allows keys with typographic errors, making the use of the map very cumbersome. A solution could for example use a globally accessible, static list of key identifiers, which could be used as a reference. Even better would be a solution that only allows those values as keys, maybe based on a key identifier enum.

The virtual buildings also have a position attribute which is used in the rendering preparation code of TableView.cpp, although the attribute is never assigned a value in the current state of the VirtualBuilding class. Instead, most of the framework rely on the positions of the building's vertices. However since hiding buildings required the center of each building, a new center attribute has been added to VirtualBuilding, which is initialized as the average of the vertices' positions. That means there are two attributes for the position/center, but to not break the hard to understand rendering code, it has not been touched. If the rendering in TableView is ever changed to not use the building positions that do not even have meaningful values, the duplicate attributes should probably be unified.

References

[1] M. Ruegenberg and P. Hirschbeck, "Interdisciplinary project (IDP): Realization of enhanced interaction for a multitouch table," 2013.

[2] J. O. Wobbrock, M. R. Morris, and A. D. Wilson, "User-defined gestures for surface computing," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1083–1092, ACM, 2009.

[3] S. Damaraju and A. Kerne, "Multitouch gesture learning and recognition system," in *Extended Abstracts of IEEE Workshop on Tabletops and Interactive Surfaces 2008*, 2008.

[4] "2014 OpenNebula Cloud Architecture Survey Results." http://netcode.ru/dotnet/?lang=&katID=30&skatID=258&artID=6761. Accessed: February 2017.

[5] H. Lü and Y. Li, "Gesture coder: a tool for programming multi-touch gestures by demonstration," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2875–2884, ACM, 2012.