Graphics & Visualization Project Report (2014)
C. R. Johnson, M. Meyer, A. Herbert-Voss (Editors)

ParaCELLsys: Interactive Tool for Hippocampus Model for Data Research

M. Martin[†1], M. Meyer[‡1], A. Herbert-Voss[§1], C. R. Johnson[¶1]

[1]Scientific Computing & Imaging Institute, SLC, Utah

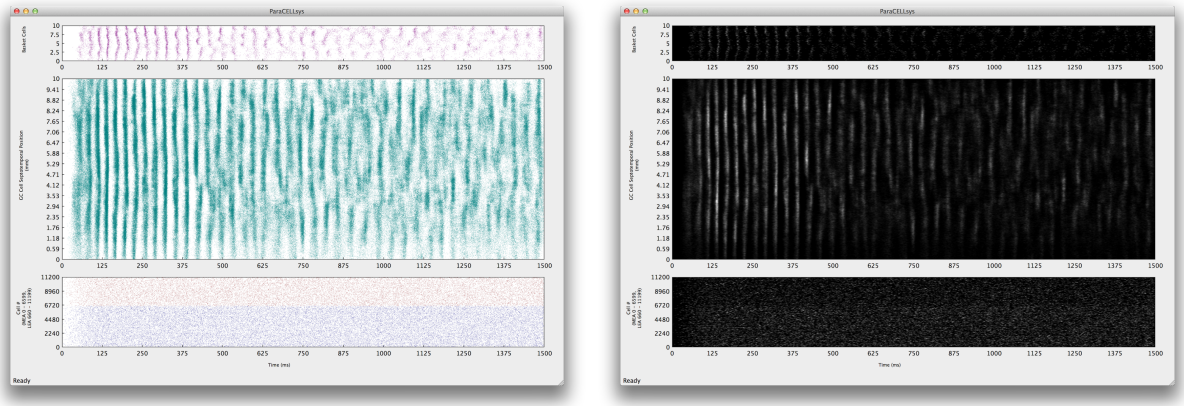

Figure 1: ParaCELLsys is a visualization user-interface offering real-time interaction for the rat model hippocampus dataset. Together, the University of Southern California(USC) and the Scientific Computing & Imaging Institute (SCI) provides visual interactivity to users. This paper shows improvements made to the initial visualization to render from 40+ seconds to under a second.

Abstract

The SCI Institute has been creating tools to visualize axons and dendrite conductivities between multiple neuron networks as well as a tool for visualizing spike data for a large set of hippocampus cells. Previous tools used by USC were non-interactive due to the large number of data points; the new tool developed by SCI, called ParaCELLsys, renders the neural simulation data in under a second (a 40-fold decrease in rendering time), providing real-time exploration and modification capabilities.

1 Introduction

Implantable medical devices have been developed to treat neurological disorders such as deafness and Parkinson's disease. The complexity of the interface portends significant challenges, and a better understanding of the interaction between the applied field and the biomolecular/neuronal network, is necessary. Better neural predictive models in terms of resolution, accuracy of topology, and bioelectrical behavior modeling will help to reach this understanding.

In ParaCELLsys, visualization strategies are used to view the interaction between large sets of electrodes communicating with the hippocampus in the brain using the rat model. The four groups we look at consist of Dentate Granule, CA3 Pyramidal, CA1 Pyramidal, and Dentate Basket cells.

† Mavin M. Martin - mavinm@sci.utah.edu
‡ Miriah Meyer - miriah@cs.utah.edu
§ Ariel Herbert-Voss - ariel@sci.utah.edu
¶ Chris R. Johnson - crj@sci.utah.edu

The University of Southern California(USC) initialized the project as a visualization tool for researchers to use in the study of cell behaviors and patterns. Unfortunately, rendering the user-interface was slow and unresponsive consuming 40+ seconds for each interaction attempt made on the rendered user-interface.

This paper explores improvements made to the original user-interface by combining hardware and software parallelization using advanced algorithms. Results show drastic speedups in performance that makes ParaCELLsys responsive in real-time. In this paper, we discuss the algorithms used, results produced, installation procedure, future work possibilities, and a walk through tutorial. In the algorithms used, we talk about how we achieved minimal modifications into making the original Python scripts verbose from an aggregated matplotlib framework to using OpenGL. To achieve real-time study of the data cells, we use the KD-Tree implementation to allow search queries to be quick and efficient. We also take advantage of CUDA cores using PyCUDA to allow a modified structure of the Grid Filter algorithm [NVI] combined with other algorithms to achieve clustering the datasets to a visible height-map field.

2 Software and Hardware Parallelization Algorithms

The project structure offers a wide variety of data types using Python's Numpy library. The dataset used in the research was provided in a binary file and extracted out using cPickle. The interactive Graphical User Interface (GUI) was implemented with a customized KD-Tree allowing fast real-time interaction. In this section, we discuss the modified algorithms used to fit the needs of allowing users to interact with the dataset in real-time.

2.1 GPU Acceleration using OpenGL

The original user-interface provided with the dataset rendered slowly and was not interactive because the number of data points being aggregated on the CPU was too large. The original Python visualization script used the Matplotlib library, which works well for small-scale visualizations but is an unsupported framework for larger projects. To take advantage of today's modern real-time interaction, we modified the original code to use OpenGL, allowing parallelization on the GPU and improving performance.

Using Python's PyOpenGL library, we changed the original code into a fast interactive environment with minimal modification. PyOpenGL converts Python objects to C, allowing our code to take advantage of OpenGL. It offers some nice tools to interact with creating matrices and arrays using Python's Numpy library. We also used PyQt's framework in helping quickly build an interactive user-interface with our code. PyQt allows us to use pre-created frames and canvases for simple integration with little code. This allows quick functionality that binds to our custom-written Python functions using PyQt's best practices coding convention.

Taking advantage of the GPU allows a huge speedup in interaction with over a million data points. There is still a large cost associated with the interactive tool due to the hardware bus that the data transfers from the CPU to the GPU every rendering cycle. To account for this, vertex buffer objects (vbo) are created when the initial rendering is initialized. The vbos are then stored on the hardware with address pointers to the memory on the GPU. This minimizes the amount of traffic being sent from the CPU to the GPU during each rendering cycle. Since vbos are only interactive as entire sets, individual cell data points needed for highlighting are sent to the hardware allow pop-outs on individual cells. To do this, the custom KD-Tree explained in Section 2.2 is used to query the individual point coordinate on the CPU before sending it to the GPU.

2.2 KD-Tree for Real-time Data Interaction

KD-Trees [Zho08] are extremely useful for querying individual points without traversing an entire dataset. The individual point queried returns the nearest-neighbor point with $O(log\ n)$ lookup time where n is the number of nodes in the tree.

In ParaCELLsys, the KD-Tree is used to lookup the closest neighboring cell to the mouse cursor position at each time frame. The Granule Cells window containing 868,708 placed into a balanced KD-Tree would take at most 20 lookups to find the nearest neighbor.

In the modified KD-Tree, the tree is adjusted to allow functionality that initially appears to be full of noise when searching for the nearest-neighbor to the mouse cursor. Figure 2 shows the reasoning for a large amount of noise in the nearest-neighbor lookup in the implementation. The white circle in the figure represents the position of the mouse cursor. Although the green dot appears closer to the cursor, it is the blue dot which is actually closer. Figure 3 shows Figure 2 in perspective with the user's visual perspective. To fix this scaling problem, we normalize all the data points into a unit field shown in Figure 4.

Another issue is initializing the creation of the KD-Tree. In simulations, the KD-Tree is usually rebalanced or recreated every frame cycle because the dataset is typically dynamic. The benefit of ParaCELLsys is that the tree only needs to be created once when initializing the first rendering and again whenever the window scale is tweaked.

Additionally, the initial view window does not have a ratio of 1:1; it instead has a ratio of 4:9, where 4 is the vertical bar length and 9 is the horizontal bar length for each cell group. When we create the tree the first time, we simply scale the window on the back end to the view ratio of 4:9. This allows a balanced KD-Tree that is in the perspective of the

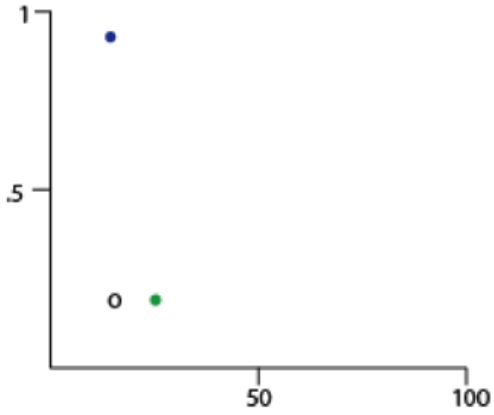

Figure 2: Assuming the unit of the vertical and horizontal axis are the same, we represent the cursor as the white circle. The view from the user's perspective creates the incorrect assumption that the green dot is closest to the cursor. The proper view, where the blue dot is correctly shown to be closest to the cursor, is shown in Figure 3.

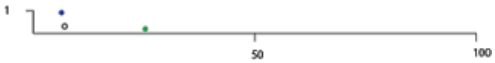

Figure 3: We scale Figure 2 to the proper view by normalizing the scaled vertical axis to the computer's perspective where the blue dot is actually closer to the cursor represented by the white circle.

viewer. When the view window is manipulated modifying the ratio, we create a new tree using the filtered dataset. In ParaCELLsys, the longest tree creation time is during program initialization because it is the only time all the data is in one frame before the user filters the dataset to view a subset. We cache the tree in a serialized object so that every time the user comes back to the original view, it doesn't have to re-create the entire tree.

When printing out the data, we denormalize the data cell position to allow retrieval of its actual location. The queried position is then returned, illustrated in Figure 5. The nearest-neighbor cell is highlighted so the user knows which cell is closest to the mouse cursor. Left-clicking presents the information of the particular cell inside of the output log window. The information presented is the distance of the cell to the cursor, the focus point's cell ID, and the actual information about the cell location in its original spatial location.

The tree also has added serialization support to cache the initial dataset on the hard drive without continually re-creating a balanced KD-Tree every time the program is launched. This speeds up the computing time for researchers who are constantly closing and reopening the program. Python supports serialization of the objects into binary files

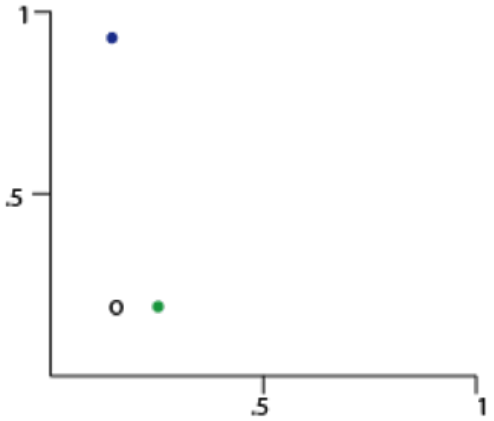

Figure 4: To account for the inaccurate nearest-neighbor lookup from the user's perspective, we normalize all the values on each axis to a unit scale. This allows a correct nearest-neighbor lookup using the mouse cursor against the data on the view window.

```
Focus Point Number= 14592
Distance from Mouse= 0.0103465919178
Point = (34.0750012547, 3.81606638432)
```

Figure 5: When selecting a particular cell that is being hovered over with the mouse cursor, the following information appears on the window output. The 'Focus Point Number' is the cell ID that was stored in the tree. The 'Distance from Mouse' shows how close the point is to the mouse in its normalized space. The 'Point' is the actual location of the point (x, y) in its spatial coordinate space.

that can be retrieved and loaded into the original class structure.

2.3 CUDA's Hardware Parallelization

Retrieving a height-map is useful when looking at clusters or the density of certain areas of a graph. The purpose of creating a height-map is to see the density of cells in each particular location. Because each cell has its own unique location, the unique cell position is turned into a continuous value that can be compared to neighboring cells. The result is shown in Figure 6.

The algorithm used to get a proper height-map makes the cells none-unique to their location. To accomplish this, a Gaussian blur to each and every cell position is done. To test and make sure it works properly, a grid of buckets are created represented by each pixel. A Gaussian sigma value is assigned to the point to ensure that it would fade as the cell is further away from the center of the bucket position. The whitest point in Figure 7 is the actual point saved in the height-map view.

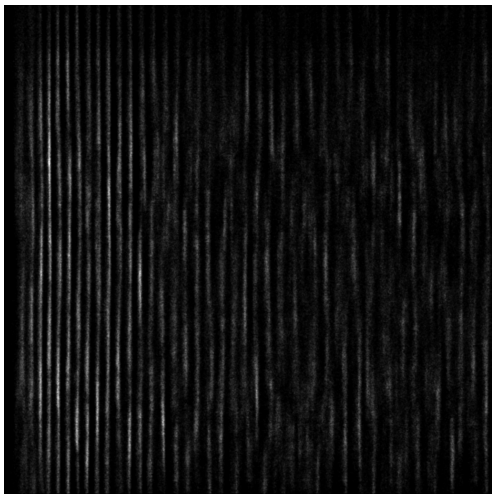

Figure 6: Output of the Granule Cells height-map within the hippocampus. With 868,708 cells, the granule cells represent the largest group of cells in the dataset. The density of cells inside of a single bucket is represented as a single pixel combining all cells in the overlapping region. Normalization then scales values from 0 to 1, where 0 is completely black and 1 is completely white in this figure.

Figure 7: Using a smoothing on a single point is used as an example to show the effects of getting a proper Gaussian blur to allow continuous integration. The value of each grid cell represented as pixels becomes normalized to a range between $[0,1]$, where 0 is black in the image and 1 is white.

Using less grid numbers in our demonstration shown in Figure 8, we discuss the algorithm used to accomplish this. A static grid size specified in the python script parameters at initialization maps the cells spatial positions on some specified Cartesian plane to the center of each grid bucket cell on the same coordinate system. For each and every grid cell $S(x,y)$, compare it to the center position of the cell $p_n(x,y)$. Grabbing the distance between the points using a Gaussian blur would output larger values closer to the point $p_n(x,y)$ and values quickly approaching zero as the distance becomes farther away as seen by looking at Equation 1. This technique is used to apply a smoothing. The integration of the values of each distance is stored inside the grid location $S(x,y)$ for every grid location as seen in Equation 2.

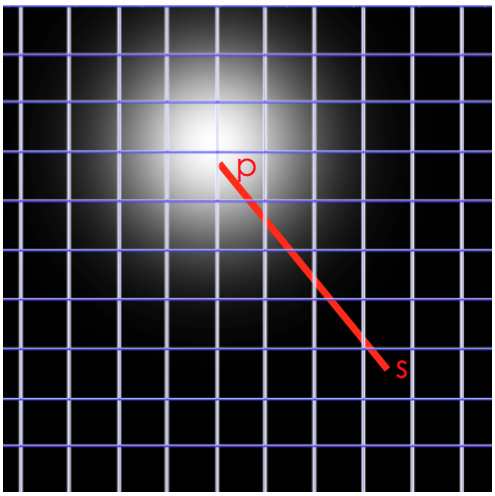

Figure 8: To achieve retrieving a continuous smoothing on each point, grid bucket centers are mapped to Cartesian coordinates on the plane where cells are located. For each cell corresponding to point $p_n(x,y)$, we iterate through each grid bucket making the center of the bucket point $S(x,y)$ and calculate the distance between $p_n(x,y)$ and s as seen in Equation 2. We increment the values of each bucket cell initiated at 0 for every $p_n(x,y)$ point giving us a summation of all the points $p_n(x,y)$ in the specified s position.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1}$$

[Sha01] [NA08]

$$S(x,y) = \sum_{i=0}^{n} G(|x - p_n(x)|, |y - p_n(y)|) \tag{2}$$

Now this is a good process that can be done on the smaller grid sizes for a small set of cells quickly. Using this technique on a large grid size or with many cells becomes a large computation. To show a bad case, lets use the grid size of 1920x1080. With all the cells provided in the three hippocampus datasets, computing the value on the buckets initialized for the 951,112 cells contained using Equation 3 by passing in $C(951112, 1920, 1080)$ =1,972,225,843,200 computations.

$$C(m,n,p) = m * n * p \tag{3}$$

Computing the Gaussian blur on the CPU with this this many computations would take days. Since each grid location is independent of neighboring grid locations, running multiple threads would compute all the grid cells simultaneously. The accumulated value of all the points in a particular grid location is then returned containing the integrated value. Additionally, using the hardware supported GPU acceleration asynchronously with the CPU takes better advantage of the computer hardware. The laptop used in the rendering process contains 384 CUDA cores that can be used in parallel with the CPU when computing the Gaussian blur used.

Python provides several CUDA libraries for GPU support. In ParaCELLsys, PyCUDA is used allowing the injection of C code directly into the python script. The C code then compiles at runtime allowing access to the CUDA cores using its direct API for supported computers. It's also integrated with Python's Numpy library making the initialization of arrays quick and simple. Additionally, the code provided by USC contained Numpy matrices causing simple integration with PyCUDA. Integrating PyCUDA into the code increased rendering time of the dataset to under a minute for a 1024x1024 grid size.

Further improvement was added to PyCUDA using the commonly used grid filtering technique. Grid Filtering [NVI] is commonly used in collision detection for real-time video games.

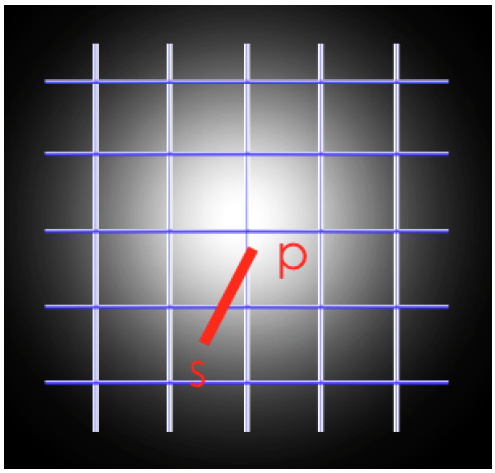

Figure 9: Taking the original grid from Figure 8, filtering is done on a subset of the original grid to ignore evaluation of the point intensity in spaces where the value is near 0.

Looking at Figure 2, the original grid is filtered into a subset grid large enough to allow the computation of the Gaussian value as seen in Figure 9. Choosing the subset size depends on the accuracy intended and correlates to the assigned σ value as shown in Figure 10.

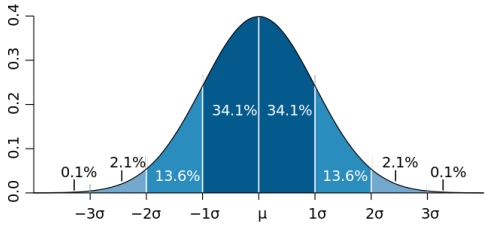

Figure 10: Using the standard deviation chart [Wik], the σ value is correlated to the smoothing σ value. Choosing $3*\sigma$ distance to move out gives a 95% accuracy.

We choose $3*\sigma$ to be the size of the subset giving between 2.1% error to 0.1% error as seen in the chart in Figure 10. Moving out to $3*\sigma$ provides 95% accuracy of the Gaussian curve. Implementing this optimization in CUDA also makes it so that grabbing a height map for the dataset goes from about a minute of computation time to a couple seconds for the rendering of the grid cells for a 1024x1024 grid size.

3 Results

The ParaCELLsys project started with goals to make the original rendering more responsive and interactive. The current state accomplishes all of the original goals. The software product is now interactive, fast, efficient, and provides useful tools for finding interesting data features that may lead to new scientific discoveries.

The original software had a 40+ second render time. ParaCELLsys has speed this up to under a second. All interactions that were in the original project took an additional 40+ seconds to re-render since Matplotlib re-aggregates all the data points to the new view for any attempted interaction. Using OpenGL, interaction in real-time contains no lag and is seamless, as shown in the screen-share recording at `http://youtu.be/tediJEFsj3I`.

The customized KD-Tree allows focusing on specific nearest-neighbor areas. Selecting the highlighted point prints out the information about the particular cell. This makes the data highly interactive using the nearest-neighbor focusing so the viewer doesn't have to be directly over a specific cell to select it. Lookup time for any particular cell takes $O(log\ n)$, where n is the number of cells in the dataset.

The latest addition to ParaCELLsys are height-maps and density clusters. Currently the height-map generation algorithm offers quick and efficient tools in studying the clusters and density deeper. The height-map is at its infant stages and has potential for further studies discussed in Section 4.

Overall, our collaborators at USC are impressed with the results and have been excited to use our tools in their

research. USC originally wanted to improve the original project with no interaction to be more responsive. Researchers are now able to interact with the data in real-time with minimal wait-time at startup and interacting with the datasets.

4 Future Work

There are wide varieties of improvements available to move the project to further heights. In this section, we discuss the potential tools that can be developed to new levels.

The dataset used shows position versus spiking-time plot of the cells. Some of the cells are re-drawn if they spike multiple times during the time cycle. Some techniques that will allow better visualization results is highlighting all of the points corresponding to the same cell id in different time locations to allow viewers to see spike-time patterns as well as the number of times a particular cell spikes during a particular time-frame. The KD-Tree can help take advantage of this by storing the cell id's pointing to arrays of all the time locations that are particular to the cell-id.

In the main window, we choose a static number of 0.2 in a range between [0,1] for transparency. This works well as seen in Figure 13. As you filter the dataset into smaller regions, it becomes much more unbearable as seen in Figure 11.

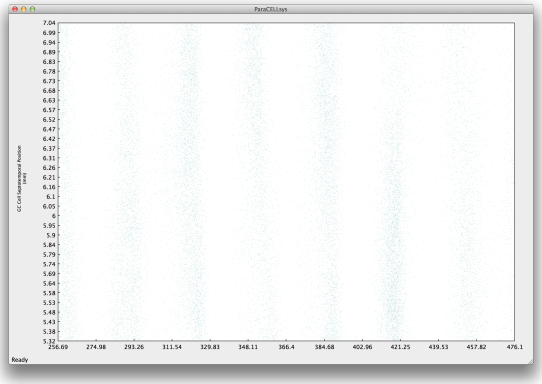

Figure 11: Showing a zoomed-in filter of Figure 13. Shown with the same 0.2 alpha value, the cell data becomes almost fully transparent due to minimal overlapping.

The height-map has great potential in providing useful tools allowing the computer to perform specific queries and logical predictions without the need of a user to manually perform specific tasks. Coloring can be added to the height-map to visually show better contrast of overlapping cells. Using two contradicting colors in the height-map like blue against yellow for values ranging from [0,1] could provide more detail. Clusters can also be arranged better by looking at buckets in the data with values larger than a certain threshold. Possible patterns in the data could be explored not be seen by the naked eye and requires a more intensive lookup by the computer's statistical analytics. Another simple tool assisting the height-map is by making values more apparent by aggregating the data values and putting it to some fractional power between [0,1]. This will scale each data bucket grid location to make all the points seem more in-tune for better viewing. Additionally, height-maps can be re-created as the user filters to a smaller subset of the data.

5 Installation Procedure

This section describes a how-to tutorial in retrieving the project and how to get the project runnable on a new local machine.

5.1 Configuring Application

To receive access to the project, please contact support@sci.utah.edu. Once given access, use Git to clone the repository from `gforge.sci.utah.edu/git/mlm/USC/ParaCELLsys`. The project is mainly done in `Python` and has code that connects to the GPU. To have OpenGL bridged properly with Python, install SIP and PyQt on the system. Make sure they are connected to Python properly. [Dun], Additionally, install all the other dependencies required to run `plot-ParallelData_pos_newFormat.py` with no errors in the output log. All required data to run the python script are included inside of `paraCELLsys/data/` directory.

If your computer has supported CUDA cores programmable via C, please install CUDA on the machine and sync with Python's library `PyCUDA`. Doing so will allow you to run the script `computeCellGaussian.py`.

5.2 Using Application

Within the `User Interface` offers several interesting tools.

5.2.1 Main Window (ParaCELLsys)

As seen in Figure 1, There are two modes with the same functionality to interact with the data. The figure to the left is the regular-view and the figure to the right is the height-map in Section 5.2.4.5.

5.2.2 Control Center

The `Control Center` allows the ability to toggle on or off different viewable windows in the hippocampus model. Simply check and uncheck the corresponding boxes to a filter that accommodates the specific assigned tasks seen in Figure 12. By selecting a subset of the checkboxes, the viewing window adapts accordingly to take full advantage of the explorer window.

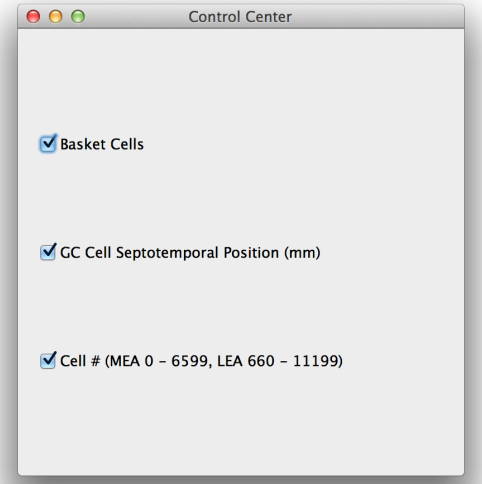

Figure 12: Controls the main widget visibilities for the viewer.

5.2.3 Menu Bar

Focusing on the main window named `ParaCELLsys`, tools in the menu bar allows the ability to toggle between the height-map view spoken in Section 5.2.4.5 and the regular-view. Toggling between the two views can also be done with the shortcut(`CTRL-Y`).

5.2.4 Main Window

The main window is the core functionality tool in ParaCELLsys. We will be mainly talking about the functionality of some useful tools inside of the main window.

5.2.4.1 Cell Transparency

After getting OpenGL to render the data, transparency to the data became a strong advantage. Transparency quickly distinguishes areas with stronger density clusters in specific regions. Between the scale range of [0,1], we choose the value 0.2 as the alpha value to help recognize regions of clustering quickly as seen in Figure 13.

5.2.4.2 Box Filter

The next thing supported is the idea of rescaling the entire dataset view using a drag technique for selection as seen in Figure 14.

By doing a box filter, the user is able to view the specific dataset region the user is interested in. To initiate the box

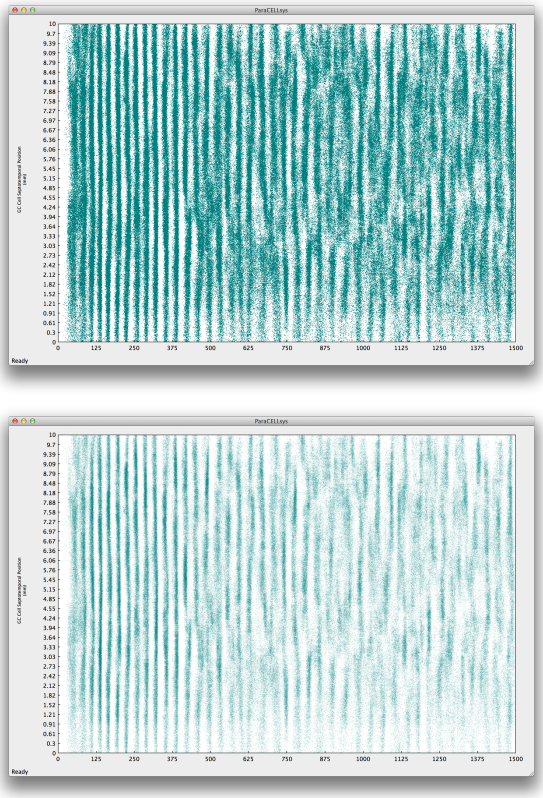

Figure 13: Seen in the image on top shows no transparency in the rendering, viewing overlapping is impossible and clusters are hard to find. In the image on bottom, we adjust the cell alpha value to 0.2 from a scale of [0,1] giving the feel of translucency. This allows the data to be easily viewed for collisions and overlapping.

filter, the user holds down the left-mouse button and drags it over the region interested in viewing and releases the button. The view is then adjusted accordingly when the user releases the left-click to reproduce a new normalized KD-Tree for nearest-neighbor lookup. To reset the filter back to the original drawing, the user right-clicks on the dataset.

When an illegal operation is performed during the box filter the user defines, an error message alerts the user explaining the nature of the error as seen in Figure 15.

5.2.4.3 Zoom-in/Zoom-out Filtering

One of the flaws of the box filter is that there is no zoom-out method using the box filter without reseting to the original view. To account for this, another zoom method is added allowing the user to zoom-in/zoom-out of the dataset in real-time. The procedure is done by holding the right-mouse button down and dragging the mouse to the right to zoom-in;

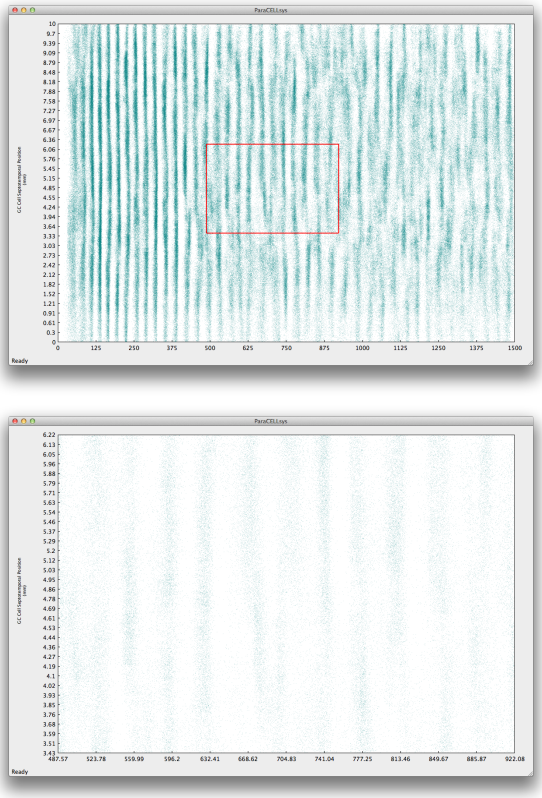

Figure 14: Seen in these two windows, users have the ability of filtering the datasets in real time. By dragging the left-mouse button, the user can visually choose the filter region in which they are interested in as seen in the top image. The bottom image shows the result of the box filter that was applied.

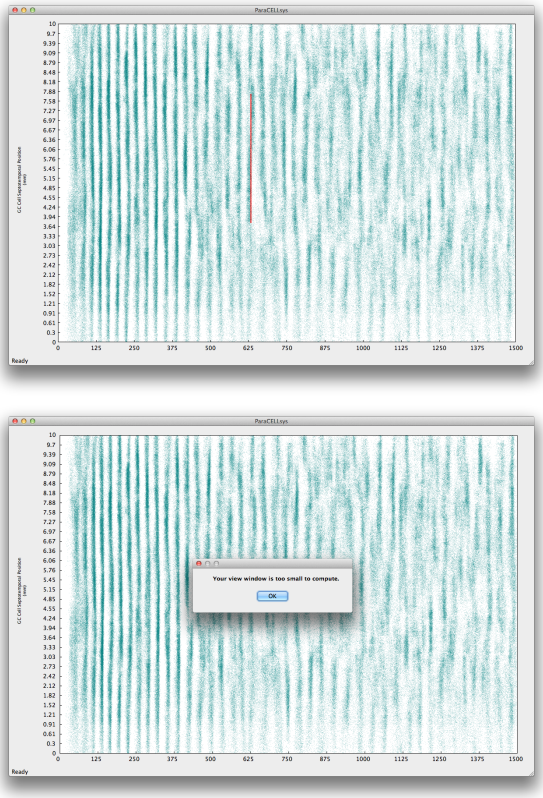

Figure 15: When attempting to do a box filter that has an illegal operation, the user is alerted with an error message regarding the nature of the actions.

likewise dragging the mouse the left would zoom-out of the dataset to see a larger view region. To represent results of this feature, we show a data region larger than the captured original view as seen in Figure 16 using our zoom-out tool.

5.2.4.4 Zoom-in/Zoom-out & Box Filtering Combined

In Figure 17 shows the combination of using the zoom-in/zoom-out tool on the data followed by initiating the box filter using the rectangular tool to zoom-in to a particular region. Using the two tools hand-in-hand allows the ability to most efficiently explore the dataset.

5.2.4.5 Normal/Height Map View

We offer two types of views inside of ParaCELLsys. The Normal view allows the user to interact with the cells as shown in most other example figures shown earlier in this paper. There is an additional view offered on the dataset called height-map visualization. Viewing the height-map performs all the same functionality with interaction as the normal-map.

The difference with the height-map tool is that an image is generated using Gaussian blurring described in Section 2.3. We then show the height-map field rather than the vertex buffer object of cell points with the same user-interactions on the re-rendered dataset. Figure 18 and Figure 19 show some of the similar interactions previously shown the normal-map view.

5.2.4.6 Shortcut-Keys

There are several tools that allow the user to quickly perform certain queries. The menu-bar has a list of the different shortcut keys that can be applied. We list the current shortcut-keys that are available in ParaCELLsys.

- `CTRL-Y` - Swap between normal/height map view

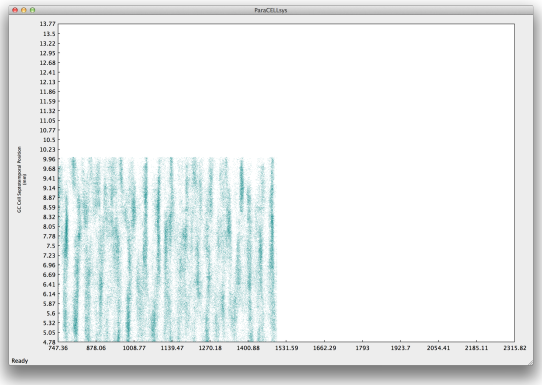

Figure 16: In this illustration, the functionality of the zoom-in/zoom-out filter is used by holding the right-mouse button down and dragging the mouse to the right zooming in. Dragging the mouse to the left will zoom-out. In this figure, illustrated show zoom-out filter result. As shown, this case represents a zoom-out filter that reaches beyond the dataset dimensions.

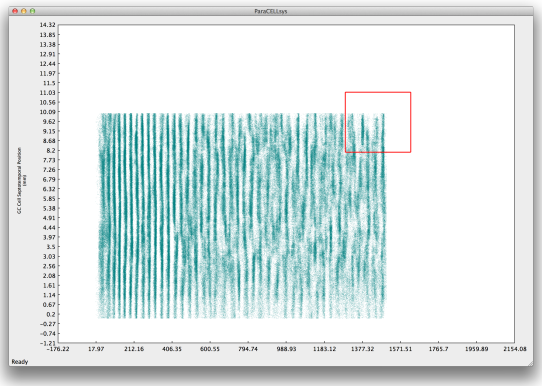

Figure 17: Showing the combination of the box filter and the customized zoom-in/zoom-out filter used conjointly to allow quick interaction with the dataset.

- `CTRL-T` - Open user Tutorial
- `CTRL-Q` - Quit application

References

[Dun] DUNN W. A.: Installing PyQt... because it's too good for pip or easy_install. http://movingthelamppost.com/blog/html/2013/07/12/installing_pyqt_____because_it_s_too_good_for_pip_or_easy_install_.html. [Online; accessed 3-July-2014]. 6

[NA08] NIXON M. S., AGUADO A. S.: *"Feature Extraction and Image Processing", page 88.* Academic Press, 2008. 4

[NVI] NVIDIA: Chapter 32. broad-phase collision detection with cuda. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html. [Online; accessed 18-July-2014]. 2, 5

[Sha01] SHAPIRO L. G. & STOCKMAN G. C.: *"Computer Vision", page 137, 150.* Prentice Hall, 2001. 4

[Wik] WIKIPEDIA: Standard deviation. http://movingthelamppost.com/blog/html/2013/07/12/installing_pyqt_____because_it_s_too_good_for_pip_or_easy_install_.html. [Online; accessed 16-July-2014]. 5

[Zho08] ZHOU K.: Real-time KD-tree construction on graphics hardware. In *SIGGRAPH Asia '08 papers*, Hart J. C., (Ed.). ACM SIGGRAPH Asia, 2008. 2

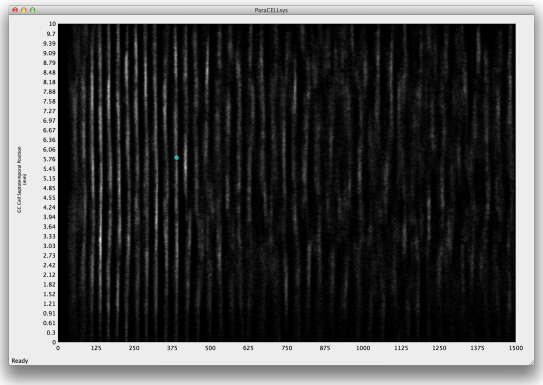

Figure 18: ParaCELLsys offers the ability to change the view-type to a height-map view. Using this mode allows the ability to find out how many cells overlap one another along with how much neighboring cells collide a given region using our modified Gaussian blur algorithm described in Section 2.3. Additionally, as seen by the blue dot in the figure, the height-map view still allows the user to take advantage of focusing on individual cells parallel to the normal-view method.

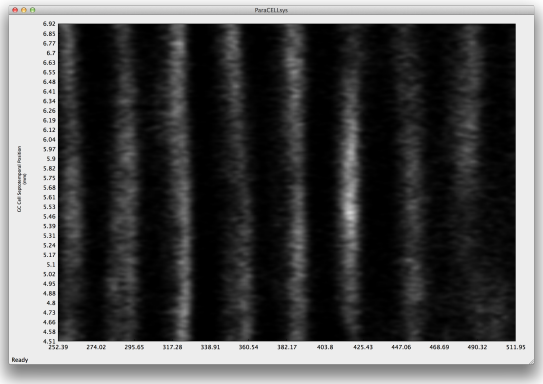

Figure 19: Showing Figure 18 after a box-filter has been applied to show the support of zooming in to the generated image and taking advantage of mipmapping.