Haptics Project Report:

Interactive 2D Collision Detection for 2DoF car rendering based on BVTT

Overview:

An interactive 2D collision detection algorithm was designed and implemented to check the collisions between the user-controlled avatar and the objects in the 2D world populated with static and dynamic objects. The objects and avatars are all represented using Axis Aligned Bounding Box (AABB) hierarchical tree structure. This representation is currently computed manually. A bounding volume traversal tree(BVTT) is dynamically created to compute the collision check on a bounding volume tree(BVT). BVTT only contains the nodes with collisions and other combinations are pruned. This BVTT traversal method reduces the computational needs and makes the algorithm fast. This project was able to design a 2D collision detector with worked in real-time and computed the collisions at a speed of ~1.38 x10^-3 secs per check.

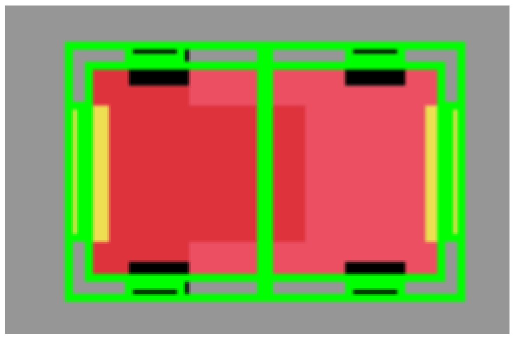
Assumptions:

- The bounding volume tree node is represented as

  : (‘parent_name’, ’node_name’, pos_x, pos_y, width, height)
- BVT nodes can have any number of child nodes as long as the bounds of the node contain the bounds of the child node perfectly.
- Objects have only two degrees of freedom: move along the x-axis and move along the y-axis. No orientations allowed.

Objects and Datastructures:

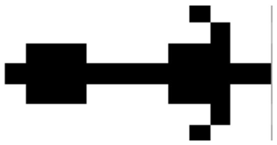
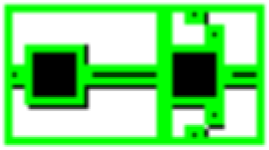
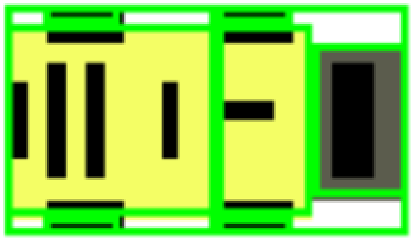
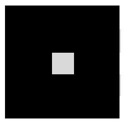
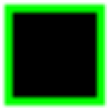
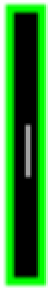
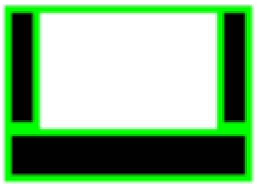
Avatar and Objects representation:



This object is represented as BVT in three levels. Top-level contains a bounding box that envelopes the avatar in a single box. The bottom level contains the bounding boxes that envelopes the elements of the avatar separately to give the most detailed representation. This level nodes are also not overlapping each other at all.

```
"avatar_car" :{
      'path': "/avatar.png",
      'size': (100,65),
      'BVH' :{
          'num_levels':3,
          'level_0':[('root',0,0,100,65)],
          'level_1':[('root','l1.1',0,0,50,65),('root','l1.2',50,0,50,65)],

'level_2':[('l1.1','l2.1',5,5,45,55),('l1.1','l2.2',0,15,5,35),('l1.1','l2.3',15,0,15,5),

('l1.1','l2.4',15,60,15,5),('l1.2','l2.5',50,5,45,55),('l1.2','l2.6',95,15,5,35),
                ('l1.2','l2.7',70,0,15,5),('l1.2','l2.8',70,60,15,5)]
      }
  }


"dynamic_truck" :{
      'path': "/dynamic_truck.png",
      'size': (105,60),
      'BVH' :{
          'num_levels':3,
          'level_0':[('root',0,0,105,60)],
          'level_1':[('root','l1.1',0,0,55,60),('root','l1.2',55,0,50,60)],

'level_2':[('l1.1','l2.1',0,5,55,50),('l1.1','l2.2',10,0,20,5),('l1.1','l2.3',10,55,20,5),

('l1.2','l2.4',55,0,20,5),('l1.2','l2.5',55,55,20,5),('l1.2','l2.6',55,5,25,50),
                ('l1.2','l2.7',80,10,25,40)]
      }
  }
```

The same approach is used to represent dynamic and static objects. This project has two dynamic objects: a truck and a bike. Also, three static objects.

Object	Bounding Boxes
	
	
	
	
	

Class structure for objects:

- Variables stored : name, pos, bvt_root, image
- Functions:
    - __init__() function
    - Draw(win) function
    - change_pos(pos)
        - Change pos and propagate the changes to the bvts
    - update_pos(pos)
        - Update the pos and propagate the changes to the bvts
    - get_pos()

Object class is used to store the objects image path, its position and its BVT root node. This class offers a *draw()* function is used to draw the object on the pygame window. *change_pos()* function is used to change the position of the object while *update_pos()* function is used to update the position of the object by the given increment. The change in position of the object is also reflected through the BVT of the object to update the bounds.

Class structure for BVT node:

- Variables stored: name, children, pos, bounds,
- Functions:
    - __init__() function
    - refresh() function
    - update_pos() function
    - draw() function

BVT node class is used to represent the nodes of the BVTs. It stores the bounds of the node, its position and its children nodes pointers. Similar to the object classes it also provides a *draw()* function and *update_pos()* function. *refresh()* function is used to refresh the bounds value based on the change in the position value. Any change made to the position or the bounds is also recursively propagated to the children nodes.

Supporting Algorithms:

make_variable_object() function:

make_static_object(), make_dynamic_object() and *make_avatar()* function are used to take the values of object from the model dictionary and create the class objects for the proper representations.

generate_BVT() function:

generate_BVT() function is used to create the tree structure based on the BVT node class objects from the model dictionary. It stores the root node in the object class.

visualise_BVT() function:

This function is used for visualizing the nodes in the BVTs.

collision() function:

collision() function is used to check the collision between two bounding boxes.

BVT_collision() function:

BVT_collision() function is used to check the collision between two BVTs. It takes two root nodes and gives the bool output. Its working is based on the bfs- tree traversal with tree pruning.

World_collision_checker() function:

World_collision_checker() function is used to check the collision between all the objects in the simulation world. It sequentially checks the collision between the objects and the avatar.

Main Algorithms:

Simulator algorithm:

- Preprocess the avatar, dynamic_obstacles, and static_obstacles into the respective class structures from the config file.
- Generate the BV Tree for them and add its roots to the class structures.
- Initialize the world in the simulator and populate the world with the models.
- While Running:
    - Check for the user input and make the changes to the avatar position.
    - Render the models on the screen.
    - Check if there is a collision between any objects in the world.
    - If collision then turns the grey background into white. And show the bounding boxes.

BVT_collision():

- If rootA and rootB are leaf nodes:
    - Return collision(rootA,rootB)
- Else if collision(rootA,rootB):
    - flag = BVTCollisionFunction(rootB,childA,flag)
- Return flag

Working:

Project Structure:

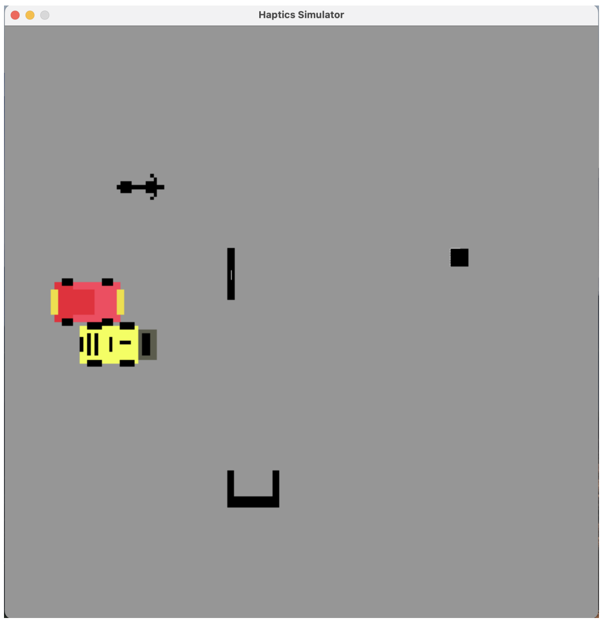
```
HAPTICS_PROJECT/
    models/
        avatar.png
        static_1.png
        static_2.png
        static_3.png
        dynamic_truck.png
        dynamic_bike.png
    src/
        utils/
            __init__.py
            colors.py
            model_configuration.py
        bounding_volumes.py
        collision_detection.py
        simulator.py
```

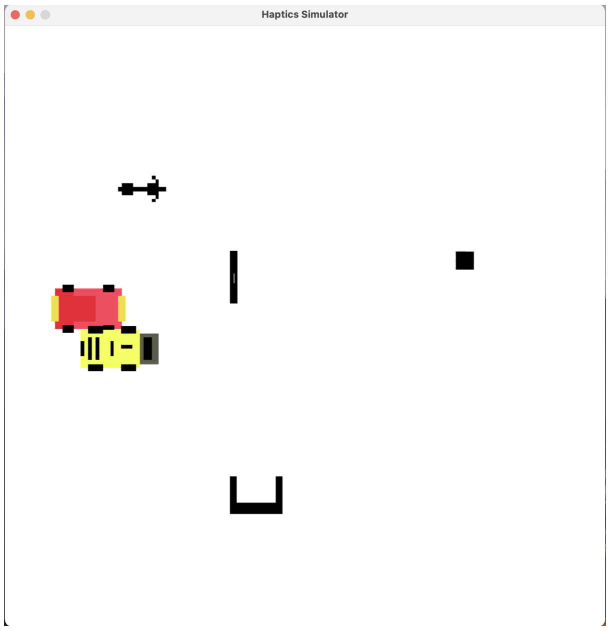
Guide:

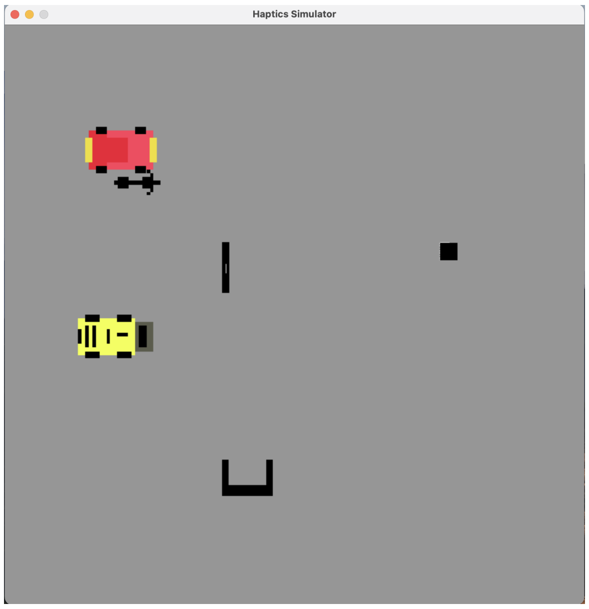
Run *simulator.py* and wait for the pygame window to pop up. This window will be titled "Haptics Simulator". Once this window pops up, a 2D world representation can be seen. Use key w to make the avatar car travel up. Use key s to make the avatar car travel down. Use key a to make the avatar car travel left. Use key d to make the avatar car travel right. The keys can be held down to perform the same action continuously, like holding the key d down will make the avatar move right with a certain speed and it will stop only once the key is let go.

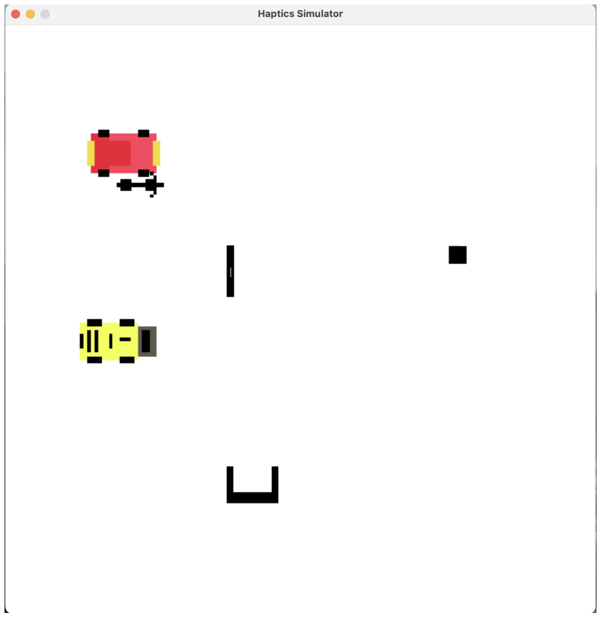
In the simulation you can interact with the world by moving towards them. If the avatar object collides with the world object then the background color changes from GREY to WHITE.

Result:


Haptics Simulator


Haptics Simulator


Haptics Simulator


Haptics Simulator

References:

[1] P. M. Hubbard, “Approximating polyhedra with spheres for timecritical collision detection,” ACM Transactions on Graphics (TOG), vol. 15, no. 3, pp. 179–210,1996.

http://graphics.stanford.edu/courses/cs468-01-winter/papers/h-apwsftccd-96.pdf.