Google Summer of Code

ANIMATION IMPROVEMENTS

GSOC 2021 PROPOSAL FOR MIT APP INVENTOR

MIT
APP INVENTOR

Proposed By: Himanshu Garg
(@Himanshu2107)

ABSTRACT and PROJECT GOALS

The animation portion of the MIT AppInventor, which includes the Canvas, ImageSprite and Ball components, currently has a lot of room for improvement. For the purpose of this project I propose to improve these aspects of MIT AppInventor:

1. Setting origin coordinates of an image sprite by the user.
2. Improving the algorithm used to detect collisions between sprites.

IMPLEMENTATION

1. Setting origin coordinates of an image sprite by the user.

Currently the origin coordinates of an image sprite are set to the top left corner. In the case of `Ball`, it is intuitive to have the origin at center, hence there exists a `CenterAtOrigin` property that implicitly lets the user decide whether the origin is at the top left corner or at the center of the ball. But in the case of `ImageSprites`, we have no idea about the contents of the image. Hence, a similar property does not make much sense. It is a desired feature that lets the user decide where he/she wants the origin to be, i.e. decide which part of the sprite do the `x` and `y` coordinates refer to. To implement this, I propose to include a draggable marker that would be a part of the component's UI in the designer. I would modify a set of properties (like `xUser`, `yUser`) in response to a drag and drop operation in the designer. Apart from having the marker as a part of the UI, I propose to add properties `OriginX` and `OriginY` in the designer. The `OriginX` and `OriginY` refer to coordinates of the unit scale from the top left corner of the sprite. That is, `(OriginX, OriginY)` of `(0, 0)` would be the top-left corner, `(0.5, 0.5)` would be the center and `(1, 1)` be the bottom-right corner. Internally we

would use u and v for these properties as they are typical in computer graphics. Whenever the user performs a drag and drop operation on the marker, the u and v coordinates would be updated. Updating the u, v coordinates would, in turn, change the position of the marker in the designer. This is similar to how the `latitude` and `longitude` properties are changed in the `Marker` component.

The idea behind using unit coordinates is that, this way the position of the origin with respect to the top left corner remains consistent on scalar changes, or changes in the image being displayed. In addition, this would allow the user to select the center, bottom-right corner or bottom-left corner of the sprite without hassle. To maintain backward compatibility, the default position of the origin (and hence the marker) would be the top left corner. To change these things programmatically, a few blocks could also be added.

Technical Details

The following discussion is on the assumption that we are extending this functionality only to ImageSprites. If the maintainers decide to include Ball, changes would be similar.

In effect we are adding two new properties to `ImageSprites`, viz. `OriginX` and `OriginY`. These refer to the coordinates of the unit scale with respect to the top left corner of the component. The draggable marker is just a way to edit these properties visually and more interactively. The draggable marker will be a part of the component's UI that calls specific methods on specific events.

To associate these properties with the component we need to follow some general steps which include:

- Add variables u and v to store the values of unit coordinates.

• To cache the values of the origin coordinates, add variables `OriginX`, `OriginY`.

• Add setters and getters for `OriginX` and `OriginY`.

• We will need to convert between the user set origin coordinates and coordinates of the top left corner. To do this add methods `xLeftToUser()` and `xUserToLeft()` (similarly for y). The use of unit coordinates would vastly simplify these methods.

Changing the origin coordinates affects many other methods in the component's class, as many calculations and changes are done according to the position of the origin. Some of these methods that may also need modifications include (non-exhaustive list):
`X()`, `updateX()`, `Y()`, `updateY()`, `moveTo()`, `PointTowards()`, `PointInDirection()`, `MoveIntoBounds()`.

Scaling the ImageSprite around different origins has different effects hence `Height()` and `Width()` would also require modifications.

As the marker is part of the component's UI in the designer, changes are also required in the `MockComponent` associated with the component.

First we will need to associate an image with the marker, which will be its visual representation. Next, we need to make that image draggable and set up appropriate events and event handlers to that image. In the event handler that handles the drop of the marker, we will note the coordinates where the marker is dropped. From these coordinates we can calculate the u and v values for that point. Next we will call
`getProperties().changePropertyValue(String PropertyName,`

`String value);` for `u` and `v` to change their values. Thus our draggable marker is implemented.

After adding the draggable marker functionality a few more changes are required to the mock component which include:

• Add variable `double u` and `double v` to store the position of origin wrt top-left corner.

• Add strings

```
private static final String PROPERTY_NAME_ORIGINX =
“OriginX”
private static final String PROPERTY_NAME_ORIGINY =
“OriginY”
```

• Override `resizeImage()` to ensure scaling is done while keeping origin consistent.

• Add methods `private void setOriginYProperty()` and `private void setOriginXProperty()`. Changes to `OriginY` and `OriginX` lead to changes in the position of the marker within the image sprite. The code that changes the position of the marker goes here.

• Update `onPropertyChange()`, `getXOffset()`, `getYOffset()`.

Now we have added the functionality to the component’s android as well as designer representation and functionality. The final step is updating the version numbers in many classes.

Apart from updating the code, updating the documentation is also important. I will dedicate some of my time to ensure that documentation is appropriately updated.

2. Improving the algorithm used to detect collisions between sprites.

The method `colliding(Sprite sprite1, Sprite sprite2)` is responsible for checking if two sprites collide. It does so by treating the sprites as rectangles and checking for overlaps. The method creates `BoundingBox` objects for both the sprites and if they intersect destructively, loops over all the points in the intersected area to check if the sprites collide. The `xLeft, yTop, Width` and `Height` attributes of the sprites are used to create the bounding boxes. In the current scenario, if the `Rotated` property of `ImageSprite` is set to true, the component would be rotated to point to the direction of the heading attribute. This is done in the `onDraw()` method of `ImageSprite.` The method does so by drawing the sprite on a canvas that has been rotated in the opposite direction. It then restores the original state of the canvas. Note that no attribute of the `ImageSprite` is thus changed. As rotating the sprite does not change the `BoundingBox` it creates (`xLeft, yTop, Width` and `Height` are not changed), the collision is detected with the unrotated `ImageSprite`, which results in vast inaccuracies and unexpected behaviour, especially in case the sprite is short and wide or tall and narrow.

There are multiple paths that we may take to solve this problem. If we decide to keep the colliding algorithm as it is, I propose the following to ensure that collision detection works properly on rotated image sprites.

1. Override the `getBoundingBox()` method in `ImageSprite.java.` Implementation does not change for Ball.
2. If `rotated` is `true`, calculate a new boundingBox for the rotated sprite using the `heading` attribute, keeping in mind that the image is rotated. We can create the axis aligned bounding box using some trigonometry.

The `BoundingBox` created for a rotated sprite, in most cases, will contain a lot of space that is not part of the sprite. This can lead to performance issues. We can optimize this algorithm using monochromatic masks for the sprites, which can lead to use of a lot less memory and hence better performance.

Another way to solve this problem is to use a different algorithm altogether for detecting collisions. There exists many algorithms in literature for this, all with their pros and cons, but I propose the SAT collision detection method for use in App Inventor. The SAT method detects collisions by projecting the shapes onto some axes and checking if there is some gap between the projections. The SAT method is meant to detect collisions in convex shapes. This can be a good choice for app inventor as it is better than using axis aligned bounding boxes for rotated image sprites and it allows us to add other sprites like triangles, squares etc, in the future, to appinventor.
There are some performance implications so the ultimate way to change the collision detection will be decided later on. We can improve the performance of collision detection using some methods. One way is to employ different methods for different sprites. For example, if we want to detect collisions between two circles, it is easier to just check the distance between their centers, than using SAT. We can add more such methods to further improve performance.

For implementation of this algorithm changes are required in the `colliding(Sprite sprite1, Sprite sprite2)` method.

Which collision detection algorithm to use is a matter of debate, and further discussion can shed light on some other possibilities.

DELIVERABLES

The proposal aims to work on the following:

1. Adding the ability for users to select the origin coordinates of image sprites using an interactive draggable marker or by changing the values of the unit coordinates in the properties panel. I aim to complete this by week 7.
2. Making collision detection work for rotated image sprites by either improving the current collision detection algorithm or by replacing it with a different collision detection algorithm. I aim to complete this by week 9. I am keeping 1 week as a buffer and to explore the glide problem.

There are some other improvements that can be addressed, but due to the reduced working hours in this year's GSOC, these are not included in the proposal.
One particular thing is the addition of a glide block, which moves a sprite to a particular position in the specified amount of time. I, if time permits, would like to explore this problem at the end of GSOC and work on it after the GSOC deadline.