

Boardmesh: Automated Polygonal Collision Geometry from Visual Inputs

Kuo-pao Yang, Jared Talbot, Luke White, Cohen Schlicher, Zachary Gros
Computer Science Department
Southeastern Louisiana University
Hammond, LA 70402, USA

Abstract— This paper presents the algorithmic foundation of Boardmesh, an open-source web API designed to automate the conversion of line drawings and object silhouettes into polygonal geometry. In physics simulations, collision shapes define the boundaries through which entities interact. While 2D simulation environments and game engines frequently rely on polygons to represent complex non-standard geometries, manually constructing these shapes remains a labor-intensive task. By automating this pipeline, Boardmesh eliminates manual shape generation, significantly accelerating development workflows while ensuring precise collision geometry for game design and physical modeling.

Keywords— *Image Processing Algorithms; Concave Polygons; Collision Detection; Physics Simulations; Video Games*

I. INTRODUCTION

Images serve as digital representations of visual information by encoding perceived light into discrete pixel color values. However, automatically extracting the geometric boundaries of individual objects within an image remains a key technical challenge that is essential for performing physics computations on image-based assets in simulation and game development environments. To address this challenge, Boardmesh provides a web service that algorithmically extracts polygonal perimeters from user-uploaded images. Leveraging these generated boundaries allows physics engines to construct interactive, simulation-ready environments directly from arbitrary visual input.

Building a perimeter requires deciding which pixels belong to the same object. This project visualizes these object boundaries directly within the image. Users can define multiple perimeters in a single frame using distinct colors that the system automatically recognizes and segments. To handle real-world lighting variations that shift pixel color values [1], we apply an algorithmic threshold for color region inclusion. Consequently, each bounded object can consist of a range of color shades rather than a single uniform color.

Many implementations of collision in physics simulations, particularly video games, will show complex sprites or images to represent an object, but the true collision shape of the object will be a primitive shape such as a rectangle or circle. This results in a rather glossy conversion of perimeter data. The current implementation of our polygon generation algorithm allows exact precision when creating the contact perimeter of an object. This superior collision detection allows for a more precise representation of physical contact on an object. The algorithm allows for representation of any digital

image whereas most algorithms only work off of drawings from the user created within the physics simulation itself. The disadvantage of this process is that it relies on the tools provided by the physics simulation software, so even if a user has trouble using these tools to draw a desired object, they have no option to utilize different software, such as dedicated software for drawing/painting, to compose necessary objects.

Our solution to this issue is to automate the process of perimeter shape creation for objects in images by algorithmically analyzing sections of preexisting images. This allows users to draw their objects anywhere, from other drawing software to real life by taking pictures of any paper or whiteboard used. There is very little existing research into two-dimensional perimeter detection algorithms; however, our current algorithm allows for no loss in image fidelity and stays true to the original perimeters provided by the pixelation.

This paper presents a particular study in the field of image processing: perimeter detection for distinct image regions. The problem is introduced and our algorithm for mapping pixels to these regions is discussed in conjunction with a unique method of polygonal shape generation. The use of these generated polygonal shapes is showcased within actual physics simulations. Finally, conclusions are presented for our algorithm, and further implementations are discussed.

II. RELATED WORK

Creating a polygonal collision mesh to accurately represent an object is already a common occurrence in a lot of physics simulators. As an example, the free physics simulator Algodoo [2] allows users to freely sketch any shape and bring it to life with accurate collision detection. This feature is also shared amongst physics-based games such as the puzzle game Crayon Physics Deluxe [3]. While these examples of giving any free-form drawing accurate collisions already exist, they notably lack a specific feature: creating a physics object with accurate collisions from a pre-existing drawing or image, rather than one that was sketched in-game.

The challenge with creating a collision mesh from an existing image stems from the fact that the pixels of an image are naturally not sorted in a way that can easily form a simple polygon. Meanwhile, a drawing of a polygonal shape that is created on the spot by a user will naturally have all of its points in the right order because as the user draws the outline of the shape with a cursor, the points of the resulting polygonal shape are being continually placed at the cursor's position in the

correct order. This eliminates the need to “sort” the points into the correct order after the user is finished drawing something.

However, there are alternative methods for accurate collision detection that do not use polygonal collision meshes at all. One of these methods will check if two objects, represented by images of a finite pixel resolution, have any pixels that overlap. If they do, then the two objects are colliding. The issue with this method is that, oftentimes, at least one of the objects in the simulation will need to sacrifice complexities in order for the simulation to maintain steady performance. These complexities include: the size of the image for an object (effectively the number of pixels in the image to check when detecting collisions), the ability for an object to be dynamically moved around according to forces, and the ability for an object to additionally be dynamically rotated around due to torque.

While the factors above are limiting for any physics simulation that wishes to feature dynamically moving and rotating rigid bodies, this pixel-overlap method still has unique use cases in simulations where many small pixel-sized objects are simulated, such as the Powder Toy [4], or simulations that feature destructible objects by simply removing pixels from an image, such as games in the Worms franchise.

We have opted to use a method that acts almost as a compromise between the two aforementioned methods—a method that uses a polygonal collision shape, allowing for dynamic movement and rotation of objects, but also allows for the use of existing images rather than having to draw them within the confines of the physics simulator. This also, hypothetically, allows for individual parts of real-world images of drawings to be used as physical objects in a simulation not dissimilar to what the Augmented Physics tool does [5].

III. IMPLEMENTATION

A. Image Simplification and Color Region Mapping

In order to create a polygonal collision shape from an image, we first need to know which pixels in the image should actually influence the shape and what pixels should be ignored and thus be considered empty space. Otherwise, every collision shape would be a rectangle with a width and height equal to the width and height of the input image.

In our approach, white and sufficiently transparent pixels are classified as empty space. Consequently, any remaining non-white, opaque pixels are grouped to form the boundaries of the polygon shape. White was chosen to represent empty space due to the prevalence of white as a background color for images, and highly transparent pixels are also empty space due to how many images that support transparency will have a fully transparent background.

For the purpose of creating a polygon from an image’s non-white pixels without respect to the actual color of the aforementioned non-white pixels, ignoring white pixels is enough to move onto the next step of the process, that being creating polygon(s) from the image. However, it may be desirable to separate the polygons created from the image based on the colors of the pixels that form them, e.g. a shape composed of only red pixels touching a shape composed of only blue pixels will create two separate polygons, rather than only one.

In order to separate pixels by color, users can simply compare the color values of each pixel, such as the values from the color channels in the RGB (red, green, and blue) color space. If two pixels have the exact same RGB values, they are clearly the same color. Otherwise, they are different colors.

This method by itself can work well enough for images synthesized by certain image painting software applications, since images containing large areas of the same exact color can be easily created. However, many other sources of images, such as real-world pictures, rarely have a sufficient number of pixels with identical colors located in the same spot. This would result in an enormous amount of polygons created due to how many different groups of same-color pixels exist in the image.

This issue can be mitigated by reducing the image’s color palette. In this approach, pixels with colors close to red are mapped to pure red, those similar to blue are mapped to pure blue, and the same process is applied to the remaining palette colors: green, black, and white. This reduction of the image’s color palette is referred to as image simplification.

This color palette of white, black, red, green, and blue was chosen due to the project’s main focus of being able to convert whiteboard drawings into polygonal shapes as well as allowing for easier use and processing of images [6]. Using image simplification on whiteboard drawings is demonstrated in Fig. 1.

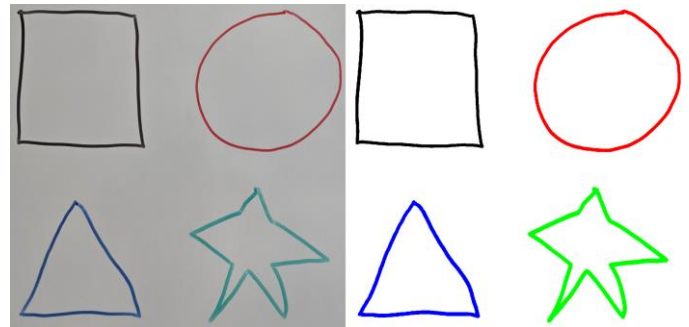


Fig. 1. A real picture of whiteboard drawings (left) and the result of image simplification after using the picture as an input (right)

As illustrated in Fig. 1, the pipeline processes the varied color shades comprising the shapes in the whiteboard image and reduces them into pure black, red, green, and blue. The algorithm employed to simplify the source image is expressed in the pseudocode shown in Fig. 2.

Excluding the color white (since it is considered empty space), there are now exactly four distinct groups of same-color pixels in the simplified image in Fig. 1, which we term “color regions.” Two pixels are in the same color region if the following are true:

1. The two pixels have the same color.
2. The two pixels are adjacent to each other, or are both adjacent to pixels that are in the same color region.

```
for pixel in image:
    c = pixel.color
    avg = (c.r + c.g + c.b) / 3
    if c.a < 10:
        pixel.color = colors.white
```

```

elif max(abs(c.r - avg), abs(c.g - avg)) < 10:
    abs(c.b - avg) < 10:
        if max(c.r, c.g, c.b) > 115:
            pixel.color = colors.white
        else:
            pixel.color = colors.black
    elif c.r > c.g and c.r > c.b:
        pixel.color = colors.red
    elif c.g > c.r and c.g > c.b:
        pixel.color = colors.green
    elif c.b > c.r and c.b > c.g:
        pixel.color = colors.blue
    else:
        pixel.color = colors.white
    
```

Fig. 2. Pseudocode to simplify an image by iterating over each pixel in the image, extracting the red, green, blue, and alpha values of that pixel's color, and setting the pixel's color to pure white, black, red, green, or blue based on those values.

These rules mean that even if two pixels are the same color, they can be in separate color regions as long as they are not adjacent to each other or are adjacent to pixels that are in the same color region. This means that color regions of the same color are often separated by either empty space (white color) or color regions of a different color. The process of assigning each of these pixels to the correct color region will be referred to as “color region mapping,” or “region mapping” for short, and an example of an image that has undergone region mapping is provided in Fig. 3. While our implementation is straightforward, more sophisticated image processing frameworks exist to address broader or more complex requirements [7].

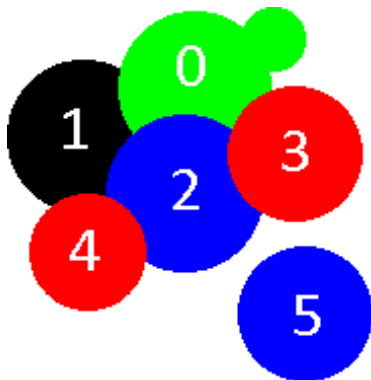


Fig. 3. An image that has undergone region mapping. The unique number for each color region is displayed on top of the color region.

Image simplification and color region mapping are important because each distinct color region corresponds to a unique shape that can be converted into a polygon. As a result, each of the six color regions shown in Fig. 3 can be transformed into a separate polygon. The polygon conversion process is discussed in the following section. Additionally, region mapping helps reduce visual noise and eliminate unnecessary shapes by removing small color regions defined as regions containing a low number of pixels, such as fewer than 50, by replacing them with white or empty space [8].

B. Color Region to Polygon Conversion and Optimization

To convert a color region into a polygon, the pixels belonging to that region must be analyzed to determine the points that define the polygon's boundary. Polygon generation can be challenging, as many different approaches exist,

including tools such as GenUDC [9]. One straightforward method is to identify all pixels along the outer boundary of a color region, specifically those that are adjacent to empty space or to pixels of a different color, and therefore are not fully enclosed within the region.

As a concrete example, consider the simplified image shown in Fig. 1. If a color such as gray is selected and applied to the top left region using a paint bucket tool in image editing software, the result would resemble the image shown on the left side of Fig. 4.

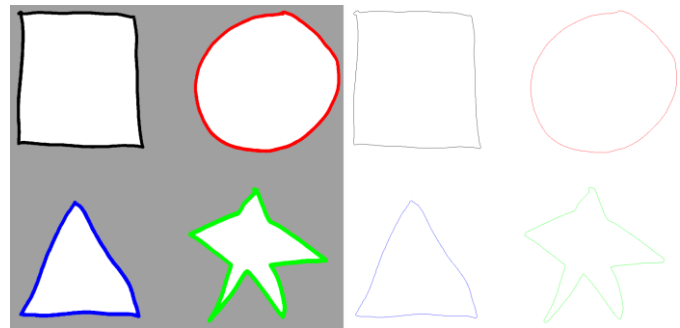


Fig. 4. The outside area of the simplified image is shaded gray (left). We can then extract every pixel that touches a gray pixel (right).

Continuing with this example, the outer boundary of a color region is identified by selecting every pixel adjacent to a gray pixel, as shown on the right side of Fig. 4. This process yields a precise, single-pixel-thin outline of the region.

Once these boundary pixels are identified, they can serve as the vertices of a polygon. However, these points must first be ordered sequentially to construct a valid, non-self-intersecting (simple) polygon. Constructing simple (non-self-intersecting) polygons is necessary in rendering and collision detection to prevent unexpected behavior and errors.

There are multiple ways to sort the points of a polygon. One method is to find the center of the polygon by adding together the position of each of the polygon's points and dividing the sum by the number of points in the polygon to find the “average position,” and then sorting each point based on their angle relative to the center point (which could be found with inverse tangent). While effective, this method is restricted to convex polygons. To accommodate concave shapes as well, an alternative approach is required.

To sort the vertices of a concave polygon, the algorithm selects an initial arbitrary pixel and sequentially traverses neighboring boundary pixels until returning to the starting point. Fig. 5 presents a variation of this traversal method in pseudocode.

However, the algorithm in Fig. 5 introduces a key structural constraint: each boundary pixel must be strictly adjacent to exactly two other boundary pixels, ensuring a deterministic sequence of predecessor and successor nodes. Adopting this constraint eliminates path ambiguity—such as resolving choice points among multiple candidate pixels, avoiding dead ends, or backtracking across previously visited nodes. Backtracking not only imposes unpredictable performance overhead but can also generate self-intersecting geometries. This path selection challenge mirrors the Traveling Salesman Problem [10], where

determining an optimal trajectory becomes non-trivial when multiple valid neighbors exist.

```

polygon = list()
previous_pixel = null

current_pixel = outside_pixels.get_arbitrary_pixel()

while current_pixel != polygon.first_element():
    polygon.append(current_pixel)
    adjacent_pixels = \
        outside_pixels.find_adjacent_pixels(pixel)
    for adjacent_pixel in adjacent_pixels:
        if adjacent_pixel != previous_pixel:
            previous_pixel = current_pixel
            current_pixel = adjacent_pixel

return polygon
    
```

Fig. 5. Pseudocode to traverse over every pixel amongst the outer pixels, adding it to an ordered list representing the points of the polygon.

To apply the traversal method in Fig. 5, each boundary pixel must possess exactly two adjacent boundary neighbors. This is achieved using a reduction process similar to the initial perimeter extraction step, systematically pruning redundant boundary pixels that contribute minimal detail to the overall shape geometry.

To identify these excess pixels, the algorithm first isolates the essential boundary pixels that must be preserved. Similar to the perimeter detection stage, this step checks for adjacency to empty space; however, rather than evaluating space outside the color region, it evaluates the previously extracted boundary pixels against empty space inside the region. This stage of the process, termed region trimming, is illustrated in Fig. 6.

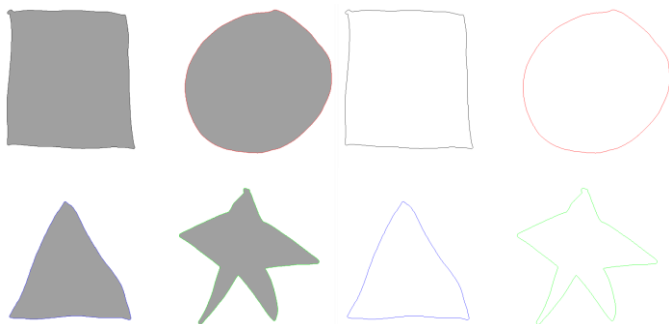


Fig. 6. The inside area of the simplified image is shaded gray (left). We can then remove every pixel that does not touch a gray pixel (right).

Among the remaining pixels shown in the right side of Fig. 6, every single pixel now has only two adjacent pixels. This means that our solution shown in Fig. 5 is now applicable. It may not immediately appear that anything has changed between the pixels from the right side of Fig. 4 and the pixels on the right side of Fig. 6, but this is due to how large the color regions used are, as well as how few excess pixels exist in these specific color regions. For clarity, there is an additional example of the process used to acquire viable pixels, this time with a smaller region, in Fig. 7.

Once the set of viable boundary pixels is identified, including those shown in the fourth quadrant of Fig. 7, and converted into polygon vertices via the traversal process shown

in Fig. 5, geometry optimization can be applied to reduce total vertex count. An excessive number of collinear vertices on a single polygon edge degrades computational efficiency, making vertex minimization essential [11]. Reducing these redundant vertices improves memory utilization and speeds up processing during physics simulations and rendering. An effective, non-destructive optimization technique removes redundant vertices along perfectly straight segments without altering the shape or geometric complexity of the polygon, as demonstrated in Fig. 8.

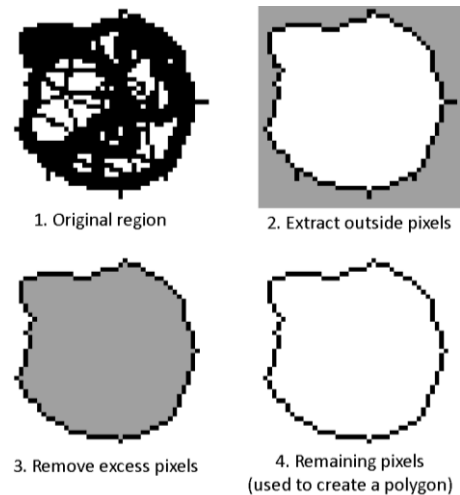


Fig. 7. The process used to extract viable pixels from a region.

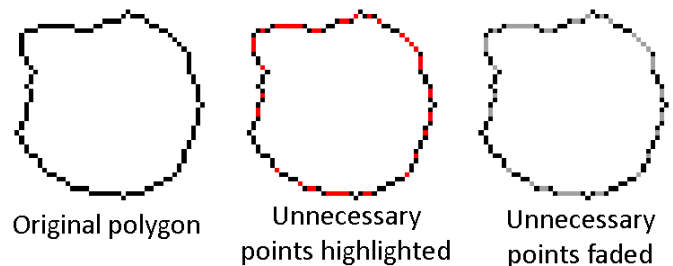


Fig. 8. Unnecessary points that can be removed without changing the effective shape of the polygon.

IV. EVALUATION

The Web API that integrates and utilizes all of these algorithms together is named Boardmesh. It uses all of the algorithms explained earlier to create suitable, color-corrected, collidable shapes for physics simulators. It is free, open-source, and anyone can make a call to it. Boardmesh can also be used to simplify an image into a restricted color palette consisting of black, white, red, green, and blue.

To demonstrate the capabilities of Boardmesh, a single-screen 2D platformer game titled Boardwalk was developed, allowing users to interact with and traverse custom images. The game incorporates the newly generated collision shapes as level obstacles derived directly from elements of the input image. These interactive obstacles are categorized into four distinct color classifications: red, green, blue, and black. Upon collision, the primary colors alter the state of the player entity,

whereas black functions as static terrain without modifying the player condition. A representative level layout is shown in Fig. 9.

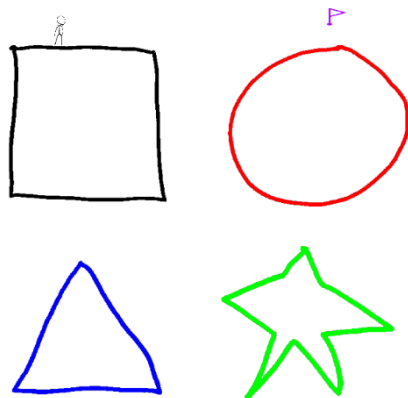


Fig. 9. An image converted to a level in Boardwalk.

Boardwalk was created using the Godot game engine. Godot is a free and open source game engine that allows for the creation of two-dimensional and three-dimensional video games. It utilizes the GDScript programming language, a language made specifically for the engine with syntax similar to Python.

Within Boardwalk, the player interacts with collision shapes in several ways, the most basic being direct contact with an object. As this is a platformer game, similar to Super Mario Bros. or Sonic the Hedgehog, the player character can run, jump, and dash across the newly created level consisting of shapes created from their image of choosing. In Boardwalk, the shapes are treated as objects in a level with the player being able to select the level's start and stop locations. As such, the ability to make contact and traverse the new shapes is necessary. As seen in Fig. 10, a player character is able to make contact and move around a newly created shape.

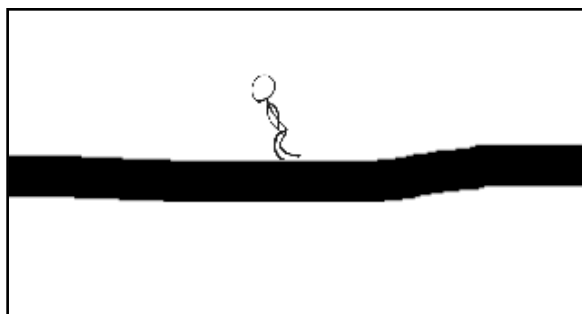


Fig. 10. The player character runs to the left across a large shape acting as terrain.

When interacting with newly generated terrain, several factors are taken into account, with the terrain's color being the most significant. Depending on what the color of the terrain is, the player's state can be altered. If the terrain is black, there are no modifications to the player's condition, as previously stated. If the terrain is red, the player will die. Upon the player's death, a small explosion effect will play and the player's location will be reset to where the player set the primary spawn point upon loading the level. Green puts the player in a sort of "dash" state where their speed is drastically increased. When dashing, the

player cannot change direction and is bound to the direction they entered the green terrain in unless a jump is performed. Upon touching blue terrain, the player will be launched upward higher than a basic jump with no limit on the number of jumps. The effects of these colors can be seen below in Fig. 11. It should be noted that while the Boardmesh Web API does take note of region color, any interactivity is strictly distinct to Boardwalk. This allows for anyone using the Boardmesh API to have the freedom to make colors interact however they desire or need them to.

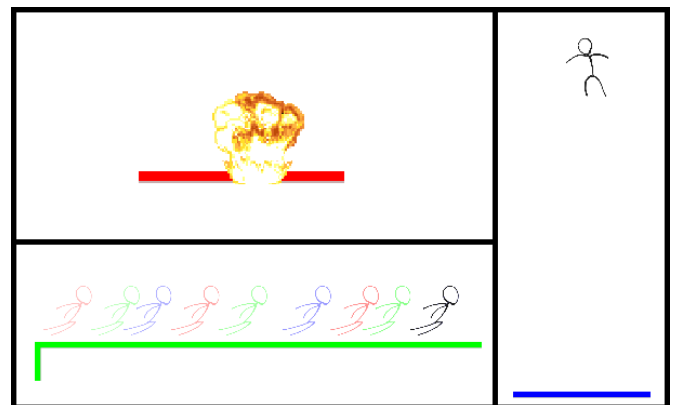


Fig. 11. The effects of the different terrain colors in Boardwalk.

While Boardwalk was made primarily for desktop use, the game is fully functional on mobile devices. By visiting the Boardmesh web page on a mobile browser, the game can be played using touch screen icons designed specifically for the game. The left and right arrow buttons are for moving the player character to the left and right, respectively. The up arrow is for jumping, and the boot with the feather activates the dash ability. The mobile implementation of Boardwalk can be seen below in Fig. 12.

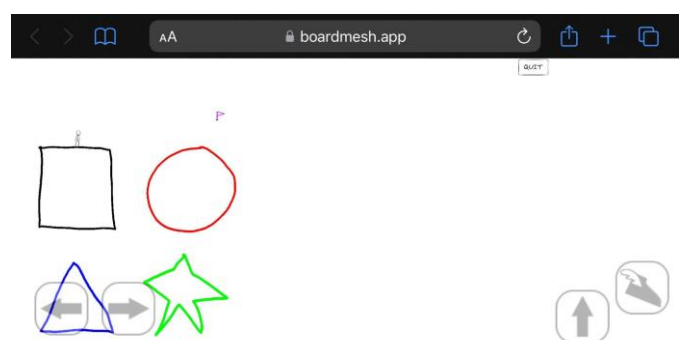


Fig. 12. Boardwalk's touchscreen icons displayed on an iOS device's web browser.

Generating polygonal collision shapes enables highly dynamic physics objects to translate and rotate with accurate collision fidelity, avoiding the computational overhead of pixel-by-pixel collision detection. While Boardwalk demonstrates static obstacles, which represent a primary application for collision polygons, it does not evaluate dynamic physics entities. To demonstrate Boardmesh in dynamic scenarios, a secondary application, Boardbox, was developed within the Godot Engine. Similar to Boardwalk, Boardbox processes user-uploaded images and converts individual color regions into rigid body physics objects with pixel-accurate

collision boundaries. Driven by simulated gravity, these objects interact, collide, and undergo mouse-driven manipulation, as illustrated in Fig. 13.

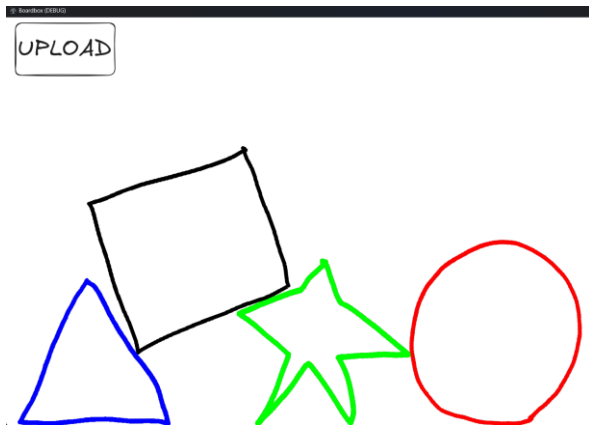


Fig. 13. An image converted into four physical objects in Boardbox, which have all fallen and collided with each other before coming to rest.

V. CONCLUSION AND FUTURE WORK

In this research, the problem of finding the perimeter of many objects within a single image was algorithmically solved with a program written in the Go programming language, Boardmesh. An application of this algorithm's output, Boardwalk, was written in GDScript, and Go and GDScript communicate using the web API hosted on boardmesh.app. Combined, there exists Boardwalk, the application of our API, and Boardmesh, the API itself.

The development of this algorithm stems from the demand for efficient image processing workflows in physics simulations and interactive software. By providing an automated method to convert source images into interactive geometry, this approach enables researchers and developers to rapidly generate custom physics simulations directly from user-provided images.

Further research may involve the inclusion of Delaunay triangulation [12], a fast polygonal processing technique, in order to create more mesh-like outputs for the purpose of simulating destructible physics objects, and the addition of more advanced color recognition. The main motivation for a new color recognition process will be due to problems with glare and shadows in photographs and improve overall image quality in order to give users a better tool to create physics-based projects and applications with.

REFERENCES

- [1] R. Raskar, A. Agrawal, C. A. Wilson, and A. Veeraraghavan, "Glare Aware Photography: 4D Ray Sampling for Reducing Glare Effects of Camera Lenses," ACM Special Interest Group on Computer Graphics and Interactive Techniques (SIGGRAPH '08). Los Angeles, California, Article 56, pp. 1–10. DOI: 10.1145/1399504.1360655, August 2008.
- [2] B. Gregorcic and M. Bodin, "Algodo: A Tool for Encouraging Creativity in Physics Teaching and Learning," American Association of Physics Teachers Physics, 55 (1), pp. 25–28, DOI:10.1119/1.4972493, January 2017.
- [3] Crayon Physics Deluxe, <http://www.crayonphysics.com>, October 2026.
- [4] Powder Toy, <https://powdertoy.co.uk>, October 2026.
- [5] A. Gunturu, Y. Wen, N. Zhang, J. Thundathil, R. H. Kazi, and R. Suzuki, "Augmented Physics: Creating Interactive and Embedded Physics Simulations from Static Textbook Diagrams," In Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), Pittsburgh, Pennsylvania, Article No. 144, pp. 1 – 12, DOI: 10.1145/3654777.3676392, October 2024.
- [6] M. A. Shantzis, "A Model for Efficient and Flexible Image Computing," In Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94), Orlando, Florida, pp. 147–154. DOI: 10.1145/192161.192191, July 1994.
- [7] S. Olsen and B. Gooch, "Image Simplification and Vectorization," In Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering (NPAR '11), pp. 65–74, DOI: 10.1145/2024676.2024687, August 2011.
- [8] Z. Li, "Low Light Image Enhancement Algorithm Based on Edge and Colour Information," In Proceedings of the 2024 2nd Asia Conference on Computer Vision, Image Processing and Pattern Recognition (CVIPPR '24), Xiamen, China, Article No. 10, pp. 1–8. DOI: 10.1145/3663976.3663988, June 2024.
- [9] R. Wang, J. Li, D. Zeng, X. Ma, Z. Xu, J. Zhang, and Q. Zhao, "GenUDC: High Quality 3D Mesh Generation with Unsigned Dual Contouring Representation," In Proceedings of the 32nd ACM International Conference on Multimedia (MM '24), Melbourne, Australia, pp. 10862–10871. doi: 10.1145/3664647.3681241, October 2024.
- [10] Z. Bo, Q. Yang, Y. Jia, X. Gao, Z. Lu, and J. Zhang, "Evolutionary Algorithm with Cross-Generation Environmental Selection for Traveling Salesman Problem," In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '24 Companion), Melbourne, Australia, pp. 635–638, DOI: 10.1145/3638530.3654166, July 2024.
- [11] P. Chen, X. Zhang, and D. Du, "SimpliGuard: Robust Mesh Simplification In the Wild," In Proceedings of the 32nd ACM International Conference on Multimedia (MM '24), Melbourne, Australia, pp. 9885–9893. DO: 10.1145/3664647.3680660, October 2024.
- [12] L. Feng, P. Alliez, L. Busé, H. Delingette, and M. Desbrun, "Curved Optimal Delaunay Triangulation," ACM Transactions on Graphics (TOG), Vol. 37, Issue 4, Article No. 61, pp. 1–16, DOI: 10.1145/3197517.3201358, August 2018.