Call for Proposals for Final Projects

Handed out Tue, Feb 21. Due Tue, Feb 28 at 11:59pm. Please submit your proposal as a pdf email attachment to Alejandro (afloresv@cs.umd.edu). If your file is too large, you can send him a link to a document he can download.

Only *one member* of each team should submit, but please include all the team members' names on the proposal.

The final projects will be due late in the semester, perhaps the last week of classes. The exact due date will be announced later.

Overview: Give a short (roughly one page) high-level synopsis of your proposed game.

Team Members: List all the members of the team. (Working alone is fine. Personally, I prefer small teams, say 2–3 people, since it is easiest to balance the workload and avoid coordination issues. If you have 4 or more people, please check with me first.)

Game Title: Proposed title, which you can change later.

General Description: The game's general structure (e.g., FPS, puzzle, RPG, single/multi-player) and the game's general "look and feel" (3-d interactive, 2-d scrolling, turn-based, etc.). What (if any) concrete games inspired your game? Please feel free to include illustrations or images.

It is not necessary to provide details at this point, and you are free to make changes in the future. (If the changes are significant, please keep me informed.)

Platform and Resources: On what system do you plan to implement/execute your game and what software tools (e.g., game engine, graphics, geometric modeling, physics, audio) or special hardware (e.g., game system, head-mounted display, or gesture recognition) will you need?

Word of advice: You and your other team members should install these systems and test that you can build and share files, models, and executables.

Coordination: How will you and your teammates coordinate your work? (What sort of meetings do you plan? Where will the source files be maintained? Do you plan to use some form of shared file storage (e.g., Github) and/or a revision control software (SVN or CVS)?

Practical Considerations: You are free to propose any general structure you like. The only pragmatic constraint is that it must be possible for you to present a short demonstration of the game to the class and to me and/or the Teaching Assistant periodically throughout the semester. Thus, if there is any special hardware needed, you will need to haul it onto campus and set it up a few times during the semester.

Tips:

Nothing is binding: You can make changes, even radical ones, throughout the semester. If you do, please keep me informed.

Previous projects: To get an idea of what is "doable," check out the videos of last semester's projects

http://www.cs.umd.edu/class/spring2016/cmsc425/final-projects.shtml

(Of course, don't be influenced too strongly. A part of your grade is based on how innovative your project is.)

Build in Layers: Start with a basic implementation so that you are confident you can successfully develop. Then, grow the project by adding enhancements and extensions. (Ambitious plans may fall apart near the end of the semester, leaving you with nothing to show.)

Do the hard things first: It is important to identify as early as possible in the development process any implementation issue that might hold up your progress. Try to determine such issues early and design a prototype to be sure that you achieve your minimum goals. Later, you can add the "bells and whistles."

Do at least one thing well: Face the fact that it is not possible to produce a AAA game in one semester. Rather than doing a so-so job on many different elements, focus instead a single concept that will make your game stand out. (This might be something internal, such as a novel technical feature. If so, part of your final demo will involve an explanation of how you implemented this feature.)

Programming Assignment 1: Unity Tilt-Maze Game

Handed out: Thu, Feb 2. **Due:** Part I is due Wed, Feb 8, 11:59:59pm and Part II is due Mon, Feb 20 at the same time. **Note the new due date!**

Late policy: Up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submission instructions will be given later.

Overview: The goal of this assignment is to learn the basics of Unity by implementing a classical wooden game called a *Tilt Maze*, where the user tilts a board in order to roll a marble through a maze.

In preparation, I suggest working through two tutorials:

Roll-A-Ball: The Unity *Roll-A-Ball* tutorial:

`http://unity3d.com/learn/tutorials/projects/roll-ball-tutorial`

Tilt Maze: A simple Unity Tilt-Maze tutorial on Youtube by *Game Design HQX*:

`https://www.youtube.com/watch?v=ma2QGWIo64Y`

There are two parts to the assignment. The first is a very simple extension of the above Youtube Tilt-Maze tutorial, and the second involves using prefabs to generate a random and dynamically changing maze.

Part-I Requirements: (Due Feb 8.)

General Structure: The basic structure of the tilt-maze board should be similar to that of the Youtube tutorial. The placement height and width of the internal walls need not be the same as in the introduction. You should at least four walls, some of which connect to each other.

Materials: Associate colors with all the objects of your scene. The ball, platform base, and walls should all have different colors. For simplicity, you can use solid colors (unlike the tutorial, you may use a solid color rather than a texture for the floor of the maze).

As in the tutorial, the walls should be associated with a physics material that results in the ball bouncing off on contact. Keep the friction very low (or even zero) so that the ball rolls easily.

Ball: You may color the ball however you like, but (unlike the walls) it should have a shiny metallic or glassy appearance.

Goal: As with the tutorial, the goal should be represented using a particle system. (You may configure the appearance of the particle system however it suits you.) In addition, there should be a "spot" on the ground that indicates where the goal is located. I represented my goal location as a very flat green cylinder.

Limited Tilt: As in the tutorial, there should be limits on how much tilt is allowed for the board.

Start and End: We will make the start and end states more complex for Part II, so you can keep it simple for now. The ball should start floating above the starting point on the board, and then fall onto the board when the game begins. (I just used gravity to do this.) When the ball reaches the goal state, it should rise up and out of the image to indicate that the game is over.

Quit/Restart: It should be possible to quit the game at any time (e.g., by hitting the 'Q' key). It should also be possible to restart it to the original state at any time (e.g., by hitting the 'R') key.

Note that when you are running the game within the Unity editor or on the web, you may not be able to quit the game. (Restart should work, however.)

You are allowed to modify these specifications (e.g., by altering the models, colors, and some aspects of game behavior), provided that your game demonstrates that you have mastered all the required elements listed above. For example, if your game uses mouse input rather than keyboard input, your player object should still be based on physics forces (as it is in the demo), and you should have some form of keyboard input (to demonstrate that you know how to do this).

- The game can be restarted by reloading the initial scene. This can be achieved by invoking UnityEngine.SceneManagement.SceneManager.LoadScene(0).
- The game can be quit with the command Application.Quit().

  Note that quitting the game while it is running within the Unity editor or a Web-based deployment does not do anything. (At least, it didn't do anything with my implementation.) However, if you produce an stand-alone executable (e.g., an ".exe" file on Windows) then quitting the game will terminate the program.
- In order to get control of the ball at the end of the game, get the ball's rigidbody component and remove it from control of the physics system by setting rigidbody.isKinematic = true. This puts the player under the full control of the program, and you design a script to control its movement.
- When you restart the game, does the lighting suddenly change (becoming much darker)? This is a common issue. For help, see the following post on answers.unity3d.com.

Part-II Requirements: (Due Feb 20.) This is mostly an extension to Part I, but there are a few modifications as well.

Ball: The ball is essentially the same as in Part I, but we reduced its radius from 0.8 to 0.5. The ball should now have the ability to "jump." When the user hits the space bar, the ball jumps vertically. (This can be done by using the function AddForce to apply a vertical impulse. Note that the direction should be *independent* of the board's current tilt.) The size of the force should be large enough so that the ball can hop over a wall. Try not to make it too high, since otherwise it is hard to control where it lands. If the ball hops off the edge then the game is lost. (Indeed, this is the only way to lose in our implementation.)

Walls: Rather than placing the internal walls in the editor, the walls will be generated by a script. The generation process is random, so that each time you restart the game, you will have a new set of walls.

Each wall is roughly the length of a cell of the grid and behaves like a swinging gate. The hinges about which the walls spins are fixed, but their orientations are random. Think of the platform as decomposed into a 10×10 grid (see Fig. 1). The grid vertices are numbered from 0 to 10 along each side (with 0 and 10 corresponding to the outer walls). The hinges of these swinging gates are placed at the *even vertices*, that is, the indices (i, j) such that $1 \leq i, j \leq 9$ and $(i + j) \equiv 0 \pmod 2$. (These are indicated with small black circles in Fig. 1). The only exception is that there is no wall placed at the center point, wich indices $(5, 5)$, since this is where the ball drops.

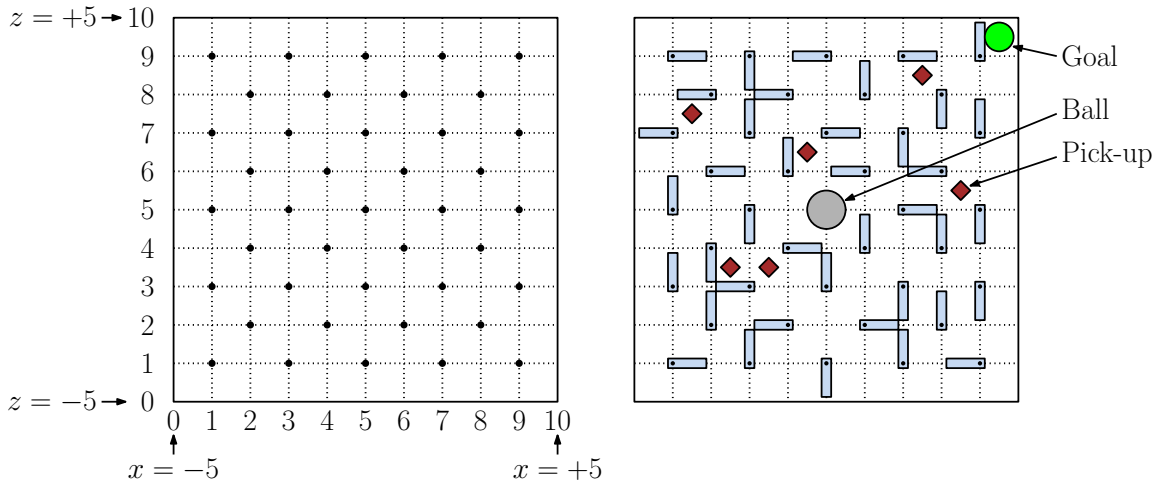


Figure 1: Grid structure.

Initially, the orientations of the walls are random, pointing either north, east, south, or west. (You may find the Unity function Random.Range(0,4) useful. It generates a random integer in $\{0, 1, 2, 3\}$. Also remember that the Instantiate command that generates a prefab can be given both the position of the object as well as its orientation in the form of a quaternion.)[1]

Throughout the game, the walls swing 90° at random times. Here is how I suggest that you to implement this. At random times, but roughly once every 10 seconds, a wall will decide that it is time to start swinging.[2] When it decides to swing, select a random direction (clockwise or counterclockwise), and initiate a process that causes the wall to execute a 90° rotation within 1 second. This can be done by applying a rotation of 90° · Time.deltaTime during each update cycle. After the rotation is complete, the wall should again be aligned with either the x- or z-axis.

Pick-ups: Similar to the Roll-a-Ball tutorial, add some pick-ups placed randomly throughout the board. The pick-ups should be centered in the middle of the squares of the grid, that is, at positions $(i + 0.5, j + 0.5)$ for $0 \le i, j \le 9$. Note that *no* pick-up should be placed on the goal position. As in the Roll-A-Ball tutorial, they should rotate (or have some other cool special effect of your own design).

The initial number of pick-ups should always be the same (we selected six) Beware of placing two pick-ups on the same grid square. Now, rather the proceeding straight to the goal, the player needs to hit some fixed number (we selected two) of the pick-ups. Please keep this number small, so that the TA can easily test your game.

Text: A new addition over Part I is that text should be displayed in your window. In the upper left, there should be a counter indicating how many pick-ups remain to be selected (e.g., "Pick-ups remaining: 2"). Note that even if the user gathers an excessive number of pick-ups, the displayed quantity should never become negative. When the required number of pick-ups have been collected, the message should indicate this (e.g., "Done! Go to the goal!"). Finally, in the upper right, there should be a message indicating whether the game has been won (e.g., "Congratulations. You

[1] Objects in Unity rotate about their center point, but each swinging wall rotates about the hinge point, which is off center. Here is a cute solution for tricking Unity into getting the rotation point over the hinge. Make your swinging wall a two-node hierarchy. The parent is an empty game object centered at the hinge, and the child is the wall, with its hinge aligned with the parent's center point. When you apply a rotation to the parent, the wall swings about the hinge as desired.

[2] This can be done by generating a random float using Random.Range (0.0f, 1.0f) and checking whether the value is smaller than Time.deltaTime/σ, where $\sigma = 10$ is the time between successive swing events. This test succeeds with probability 1/10 per second, and therefore the wall swings approximately once every 10 seconds.

win!") or lost (e.g., "Sorry. You lose!"). As mentioned above, the only way to lose the game is to jump over the wall. We detected this when the y-coordinate became a sufficiently small negative number.

Additional Notes:

Final Submission: Final submission instructions will be posted on Piazza. (If you are ready to submit and do not see the instructions, please remind me.)

Sample Executable: I usually post a sample executable of my program on the Class Projects Page. I will make an announcement about this soon.

Programming Style: While we encourage clean programming structure, good structure is not an essential part of your grade. We reserve the right to deduct points for programs that are so poorly documented or organized that the grader cannot figure out how your program is working.

Optional Elements for Extra Credit: You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your Readme.txt file. The number of points of extra-credit credit will be left to the discretion of the grader.

External Resources: If you make use of any external resources in your program (or things that you developed prior to this class), even if you modified them, you *must* credit them in your Readme.txt file. Failing to do so will be considered an act of plagiarism. If you are unsure, check with me.

Homework 1

Handed out Tue, Mar 7. Due at the start of class Thu, Mar 16. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

This homework is intentionally structured in a similar manner as the First Midterm Exam. Of course, the problems on the exam will be different, and the length and technical difficulty of the Midterm will be different.

Problem 1. Short answer questions. Unless otherwise specified, explanations are not required, but may be provided for partial credit.

(a) Suppose that a Unity game object is declared to be *static* (by checking the "Static" checkbox in the editor). Which of the following optimizations can Unity perform as a result? (Indicate True or False for each.)

(i) Navigation computation and physics can be optimized (because the object's position is fixed).

(ii) Fewer method calls are needed, because the methods Update or FixedUpdate are not called on static objects.

(iii) Some global lighting computations can be precomputed.

(iv) Space is saves because all instantiations of a static object refer to the same (shared) game object.

(b) What does it mean when we say that the vector dot product (or generally any inner product) is *bilinear*? (Express your answer as one or more vector equalities.)

(c) You have a long, thin object (e.g., an arrow) that can be oriented arbitrarily in space. Which of the following collider shapes would NOT be a good choice to represent this object (Select all the apply). Briefly *explain* your answers.

(i) Axis-aligned bounding box (AABB)

(ii) General (arbitrarily oriented) bounding box

(iii) Bounding sphere

(iv) Capsule

(d) Consider the following two computational tasks that arise in animation processing:

Task I: Given the placement of a skeletal model in a scene and an assignment to its joint angles, determine the position of a point of the model (e.g., the tip of the index finger) relative to the scene's coordinate system.

Task II: Given the placement of a skeletal model in a scene and the desired position of a given point of the model (e.g., the tip of the index finger should be touching a light switch), determine how to set the joint angles to achieve this desired result.

(i) One of the above tasks is called *forward kinematics* and the other is called *inverse kinematics*. Which is which?

(ii) Which of these two tasks is computationally more challenging? Briefly justify your answer.

Problem 2. Your new 3-dimensional game involves frisbee throwing. You need to implement an efficient collider that will (roughly) represent the shape of a flying disk. You have chosen to model the frisbee collider as a simple flat circular disk in three dimensional space. The collider is specified by three parameters: (1) the center point $p = (p_x, p_y, p_z)$ of the collider, (2) a unit-length normal vector $\vec{u} = (u_x, u_y, u_z)$ that points in the direction perpendicular to the plane on which the disk lies, and (3) a positive real r that indicates the radius of the disk (see Fig. **??**(a)). The disk has zero thickness.

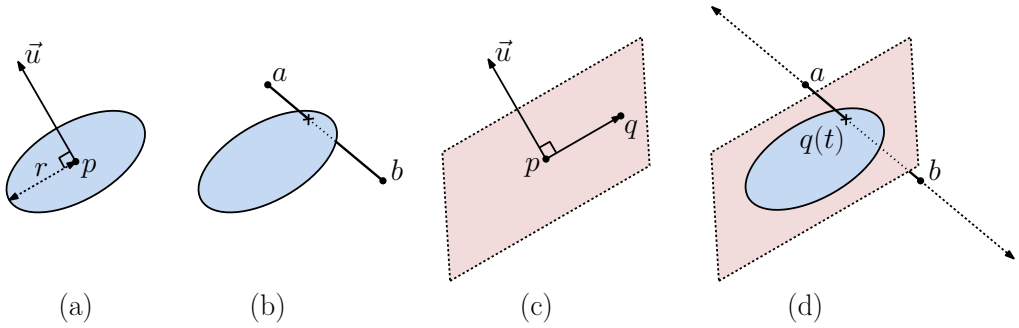


Figure 1: Problem 2.

The objective of this problem is to derive a procedure that, given a frisbee collider $\langle p, \vec{u}, r\rangle$ and a line segment $\overline{ab}$, where $a = (a_x, a_y, a_z)$ and $b = (b_x, b_y, b_z)$, determines whether the frisbee collider intersects the line segment (see Fig. **??**(b)). You may assume that $a \neq b$ and neither of the points a or b lies on the plane that contains the collider.

(a) The first step is to determine the equation of the infinite plane containing the collider disk. A point $q = (x, y, z)$ lies on the plane if and only if the free vector directed from p to q is perpendicular to the vector $\vec{u}$ (see Fig. **??**(c)). Use this fact to derive the equation of the plane. (Hint: The plane equation can be expressed in the form $\alpha x+\beta y+\gamma z+\delta = 0$ for some scalars α, β, γ, and δ. Derive the values of these four scalars as a function of the coordinates of p and $\vec{u}$.)

(b) We showed in class that any point on the infinite line $\overleftrightarrow{ab}$ can be expressed as the affine combination $(1 - t)a + tb$, for some real t. Using your answer from part (a), derive a procedure (in mathematical notation) for computing the value of t where the infinite line hits the collider plane. Let's call this point $q(t)$ (see Fig. **??**(d)). Also, present a test to determine whether $q(t)$ lies within the (finite) line segment $\overline{ab}$.

(c) Your answer to (b) should involve division by a quantity that depends on the inputs. Under what conditions (as a function of $\vec{u}$, p, a, and/or b) would the divisor(s) be equal to zero? Does the problem description exclude this possibility? (If not, what additional assumptions need to be added?)

(d) Assuming that $q(t)$ (from part (b)) exists and lies within the line segment $\overline{ab}$, explain how to determine whether $q(t)$ lies within the collider disk of radius r (see Fig. **??**(d)).

Problem 3. You are implementing a football game, and you want to simulate the process of a quarterback throwing the ball to a pass receiver. The receiver is running horizontally across

the field at a fixed speed of s_p feet per second, and the quarterback throws the ball at a fixed speed of s_q feet per second. (You may assume that both of these quantities are positive.) The quarterback needs to adjust the angle at which the ball is thrown (thus, leading the receiver) so that the ball arrives at the same time as the receiver.

To simplify matters, let us do this in the 2-dimensional plane. Assume that the quarterback is located at a point $q = (q_x, q_y)$, and at the instant the ball is thrown the receiver is at point $p = (p_x, p_y)$ directly above q. Thus, $p_x = q_x$ and $p_y > q_y$ (see Fig. **??**(a)). Let $\ell_q = p_y - q_y$ denote the initial distance between the quarterback and receiver, and assume that the receiver moves horizontally to the right.

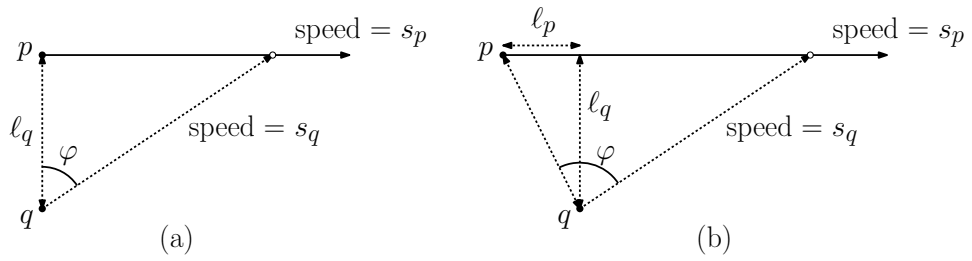


Figure 2: (a) Problem 3 and (b) Challenge problem.

Derive (in mathematical notation) a procedure, which given q, p, s_q and s_p, outputs the angle $\varphi > 0$ of the direction (relative to the vector from q to p) at which the quarterback should throw the ball so that the receiver and ball arrive at the same time in the same place (assuming that they move at their given speeds). You may express φ either in radians or degrees.

In order for your solution to exist, what assumptions need to be made about the relationship between s_q and s_p?

Problem 4. We wish to perform a rotation of θ degrees about a unit vector $\vec{u} = (u_x, u_y, u_z)$ using a quaternion representation (see Fig. **??**). We will apply this rotation to a point $p = (p_x, p_y, p_z)$. Throughout, when asked to present a quaternion, present it either as a 4-element vector or as a pair consisting of a scalar and a 3-element vector.

(a) As a function of $\vec{u}$ and θ, express this rotation as a *unit quaternion* $\mathbf{q}$. (You may express $\mathbf{q}$ as a 4-element vector or in the form (s, v), where s is a scalar and v is a vector.)

(b) Express the point p as a *pure quaternion*, denoted $\mathbf{p}$.

(c) As a function of $\vec{u}$ and θ, express $\mathbf{q}^{-1}$. (Hint: The inverse of a rotation is a rotation by the negation of the angle.)

(d) Let p' be the image of p under this rotation. Explain how to obtain the coordinates of p' in terms of quaternion operations on $\mathbf{q}$ and $\mathbf{p}$. (I am looking for a couple of short formulas. You do *not* need to expand out all the terms of the quaternion multiplication.)

Problem 5. Consider a skeletal model of an arm holding a sword in 2-dimensional space. Suppose that the bind pose is as shown in Fig. **??**(a), with the arm and sword extending horizontally to the right of the shoulder. The shoulder, elbow, hand, and tip of sword coordinate frames

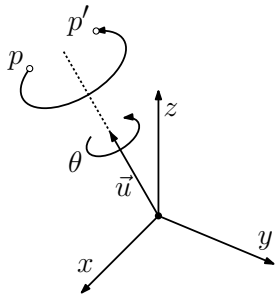
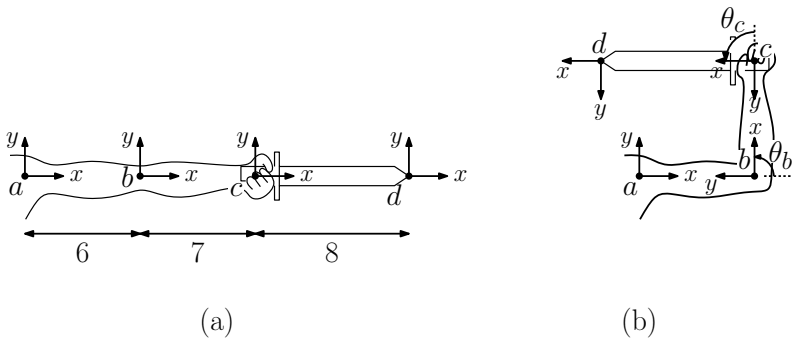


Figure 3: Problem 3.

are called a, b, c, and d, respectively. It is 6 units from the shoulder to the elbow, 7 units from the elbow to the hand, and 8 units from the hand to the tip of the sword.



(a) (b)

Figure 4: Problem 5.

(a) Following the naming convention for the local pose transformations (given in Lecture 9) express the following local pose transformations a 3×3 homogeneous matrices. (In all cases assume the arrangement shown in Fig. **??**(a).)

(i) $T_{[c \leftarrow d]}$, which translates coordinates in the sword tip frame to the hand frame.

(ii) $T_{[b \leftarrow c]}$, which translates coordinates in the hand frame to the elbow frame.

(iii) $T_{[a \leftarrow b]}$, which translates coordinates in the elbow frame to the shoulder frame.

For example, the transformation $T_{[c \leftarrow d]}$ should transform the column vector denoting the tip of the sword relative to the tip-of-sword frame coordinate (as the origin) to its representation relative to the hand frame coordinates (as lying 8 units along the x-axis). That is,

$$T_{[c \leftarrow d]} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \\ 1 \end{pmatrix}.$$

(b) Show that by multiplying these matrices together in the proper order, we obtain a matrix $T_{[a \leftarrow d]}$ that maps a point in the tip-of-sword frame to the shoulder frame. For example,

because the tip lies 21 units to the right of the should, we have

$$T_{[a \leftarrow d]} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 21 \\ 0 \\ 1 \end{pmatrix}.$$

(c) Give the following inverse local pose transformations:

(i) $T_{[d \leftarrow c]}$, which translates coordinates in the hand frame to the sword tip frame.

(ii) $T_{[c \leftarrow b]}$, which translates coordinates in the elbow frame to the hand frame.

(iii) $T_{[b \leftarrow a]}$, which translates coordinates in the shoulder frame to the elbow frame.

(Hint: You can exploit the simple structure of the matrices in part (a) to avoid the need for general matrix inversion.)

(d) Suppose that we apply a rotation by angle θ_b about the elbow and θ_c about the hand. (These are both $90° = \pi/2$ in Fig. **??**(b), but they can be any angle, positive or negative, in general.) Assume that $\text{Rot}(\theta)$ denotes a 3×3 rotation matrix, that is

$$\text{Rot}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Let's assume that all points are represented in the shoulder frame. Following the example in Lecture 9, derive a matrix (which you may express as the product of a sequence of matrices) that maps a point representing the tip of the sword in the bind pose to its rotated position. For example, in the particular case where $\theta_b = \theta_c = 90°$, this would map the vector $(21, 0, 1)$ to $(-2, 7, 1)$. (Your answer should work for any values of θ_b and θ_c.)

Explain how you derived your answer.

Challenge Problem 1. Challenge problems count for extra credit points. These additional points are factored in only after the final cutoffs have been set, and can only increase your final grade.

Generalize Problem 3 so that, rather than starting immediately above q, the point p starts at a distance of ℓ_p to the left of the vertical line passing through q (see Fig. **??**(b)). Let $\ell_q = p_y - q_y$ denote the vertical distance between q and p's line of travel. As before, the receiver moves to horizontally to the right. Now, φ is defined relative to the vector between q and p. (If you prefer, you can compute φ relative to the vertical, as before. Just be sure to explain which you are doing.)

Challenge Problem 2. Work through Problem 4, but where $\vec{u} = (0, 0, 1)$, $\theta = 90°$, and $p = (1, 0, 2)$. In addition to parts (a)–(d), also add a part (e) where you explicitly compute the coordinates of p' after the rotation.

CMSC 425 : Spring 2017 Dave Mount

Midterm Exam

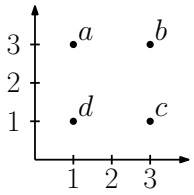
This exam is closed-book and closed-notes. You may use one sheet of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (25 points, 3-8 points each) Short answer questions. Explanations are not required, but may be given for partial credit.

(a) Give two examples that might arise in a game implemented in Unity, one where you want a rigid-body to have a *collider* and one where you want a *trigger*.

(b) You have three points p, q, and r in the plane. You want to compute a point that lies close to the center of this triangle (I don't care exactly where). Explain how to compute such a point using the operations of affine geometry.

(c) Consider the four points $a = (1, 3)$, $b = (3, 3)$, $c = (3, 1)$ and $d = (1, 1)$ in the figure below. List these points in *Morton order* (and briefly how you got your answer).

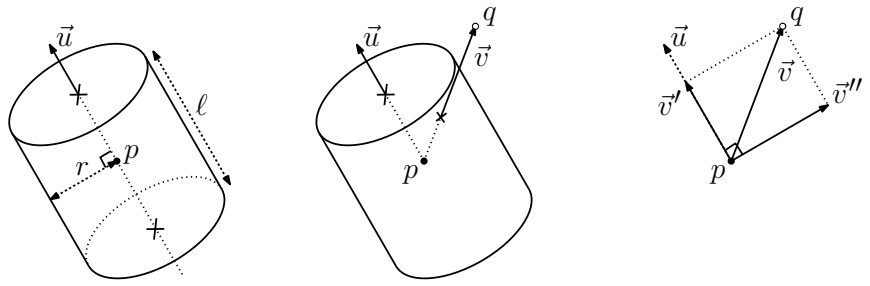


(d) In Unity, you want to rotate an object through 270° degrees to occur smoothly over a period of 3 seconds. In your Update function, how many degrees of rotation should you apply? (Hint: Use the value of Time.deltaTime.)

(e) From the perspective of performance (time and/or space) list one *advantage* and one *disadvantage* of using a hashmap rather than an 3-dimensional array to represent a grid decomposition of 3-dimensional space.

Problem 2. (25 points) The objective of this problem is to derive a test for a cylindrical collider. The collider is defined by four parameters (see Figure 1(a)):

- the center point $p = (p_x, p_y, p_z)$ of the collider
- a unit-length vector $\vec{u} = (u_x, u_y, u_z)$ that points along the central axis of the cylinder
- a positive real r that indicates the radius of the cylinder (perpendicular to the central axis)
- a positive real ℓ that indicates the length of the cylinder along its central axis



(a) (b) (c)

Figure 1: Cylinder Collider.

Our objective is to derive a procedure that will determine whether a given point $q = (q_x, q_y, q_z)$ lies within the collider (see Figure 1(b)).

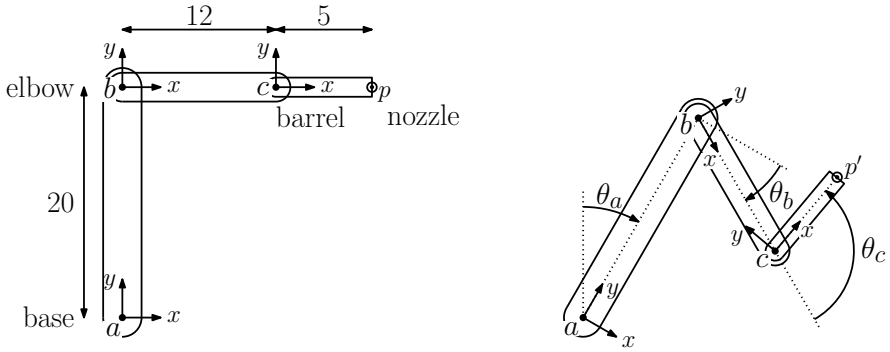
(a) (5 points) Given the points p and q, show (using mathematical notation) how to compute the coordinates of a vector $\vec{v} = (v_x, v_y, v_z)$ that is directed from p to q (see the figure (b)).

(b) (10 points) Given your answer to (a), show (using mathematical notation) how to decompose $\vec{v}$ as the sum of two vectors $\vec{v}'$ and $\vec{v}''$ such that $\vec{v}'$ is parallel to $\vec{u}$ and $\vec{v}''$ is perpendicular to $\vec{u}$ (see Figure 1(c)). (Hint: Use the dot product.)

(c) (10 points) Given your answer to (b), show (using mathematical notation) how to compute the lengths of the vectors $\vec{v}'$ and $\vec{v}''$ and then use these lengths together with r and ℓ to determine whether q lies within the cylinder collider.

Problem 3. (30 points, 5–10 points each) Your company's latest game involves a water cannon, which is used to extinguish fires in burning buildings. We will consider the problem in 2-dimensional space. The cannon's bind pose is shown in Fig. 2(a). It consists of three rotatable joints: the base, the elbow, and the barrel. Water comes out from the nozzle point p.

- Joint a (base joint) is at the origin
- Joint b (elbow joint) is 20 units above the origin
- Joint c (barrel joint) is 12 units to the right of the elbow joint
- Point p (nozzle) is 5 units to the right of the barrel joint



(a) (b)

Figure 2: Water cannon.

Given the three joint angles θ_a, θ_b, and θ_c, we want to determine the location of nozzle point p' (see Fig. 2(b)).

(a) What are the coordinates of the nozzle point p in the bind pose relative to each of the following coordinate systems? Express each answer as a 3-element homogeneous vector:
   (i) Barrel frame: $p_{[c]} =$
   (ii) Elbow frame: $p_{[b]} =$
   (iii) Base frame: $p_{[a]} =$

(b) Express the following local-pose transformations as homogeneous 3×3 matrices. (In all cases assume the bind pose shown in Fig. 2(a).)
   (i) $T_{[b \leftarrow c]}$ (barrel-frame coordinates to the elbow-frame coordinates)
   (ii) $T_{[a \leftarrow b]}$ (elbow-frame coordinates to the base-frame coordinates)

(c) What is the transformation $T_{[a \leftarrow c]}$ (barrel-frame coordinates to base-frame coordinates)? You may give your answer as a single 3×3 matrix or the product of matrices.

(d) Express the following inverse local-pose transformations as homogeneous 3×3 matrices (again, assuming the bind pose shown in Fig. 2(a).)
   (i) $T_{[c \leftarrow b]}$ (elbow-frame coordinates to the barrel-frame coordinates)
   (ii) $T_{[b \leftarrow a]}$ (base-frame coordinates to the elbow-frame coordinates)

(e) Suppose that we apply a rotation by angle θ_a about the base joint, θ_b about the elbow joint, and θ_c about the barrel joint. Let $\mathrm{Rot}(\theta)$ denote a 3×3 homogeneous rotation matrix, that is

$$\mathrm{Rot}(\theta) \;=\; \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Present a formula (as the product of matrices) that maps p in the bind pose to its position p' as a result of the rotations. Assume that p and p' are both represented relative to the *base frame*. That is, present a matrix M (as the product of matrices) such that $p'_{[a]} = M\, p_{[a]}$. (Hint: It will be faster for you and easier for me if you express your matrices by name, e.g. "$T_{[b \leftarrow c]}$" rather than as a 3×3 matrix.)

Do only one of problems 4 or 5.

Problem 4. (20 points) Extending the water-cannon problem, we want to develop a targeting tool that determines where the water will hit a vertical wall. Suppose that the nozzle point of the water cannon is located h units above the ground, and the water is being shot with velocity given by the vector $\vec{v}_0 = (v_{0,x}, v_{0,y})$. The wall is located ℓ units in front of the cannon (see Fig. 3).

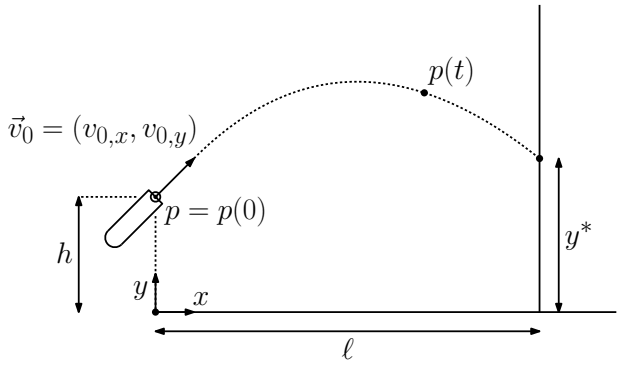


Figure 3: Water cannon, continued.

Suppose we turn on the water at time $t = 0$. After consulting a standard textbook on Physics, we are reminded that gravity results in an acceleration of $g \approx 9.8m/s^2$, and after t time units have elapsed, the position of a projectile shot at velocity $\vec{v}_0$ is given by $p(t) = (x(t), y(t))$, where

$$x(t) \;=\; v_{0,x}t \quad \text{and} \quad y(t) \;=\; h + v_{0,y}t - \frac{1}{2}gt^2.$$

As a function of h, ℓ, g, and $\vec{v}_0$, explain how to compute the height y^* at which the water hits the wall. You may assume that the velocity is high enough that the water will reach the wall. (Hint: Start by computing the time it takes to reach the wall.)

Problem 5. (20 points) Suppose that we wanted to perform a rotation of $\theta = 60^\circ$ degrees about a unit vector $\vec{u} = (\frac{1}{3}, \frac{2}{3}, \frac{2}{3})$ using a quaternion representation (see Fig. 4).

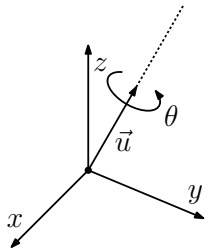


Figure 4: Quaternion rotation

(a) As a function of $\vec{u}$ and θ, express this rotation as a *unit quaternion* $\mathbf{q}$. (You may express $\mathbf{q}$ as a 4-element vector or in the form $(s, \vec{u})$, where s is a scalar and $\vec{u}$ is a 3-element vector.) Recall that

$$\sin 60^\circ \;=\; \cos 30^\circ \;=\; \frac{\sqrt{3}}{2} \qquad \text{and} \qquad \cos 60^\circ \;=\; \sin 30^\circ \;=\; \frac{1}{2}.$$

(These are the only trig values you might need.)

(b) What is the product of the following two quaternions? $\mathbf{q}_1 = (1, 2, 0, 0) = 1 + 2i$ and $\mathbf{q}_2 = (0, 3, 4, 0) = 3i + 4j$. Recall the rules of quaternion multiplication:

$$i^2 = j^2 = k^2 = ijk = -1 \qquad ij = k,\; jk = i,\; ki = j.$$

Programming Assignment 2: Elvis vs. Grannies

Handed out: Tue, Apr 11. **Due:** Wed, Apr 26, 11:59pm. Late policy: up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. The submission procedure will be the same as in the first programming assignment (see below).

Overview: The objective of this assignment is to learn more about Unity through the design of a primitive game that involves controlling the movement of characters through the use of Unity animator controllers and navigation meshes. The game consists of the following required elements:

Scene: The scene consists of Elvis's groovy floating mansion in the sky, which consists of a number of shiny golden boxes with a few pieces of purple velour furniture (see Figs. 1 and 2).

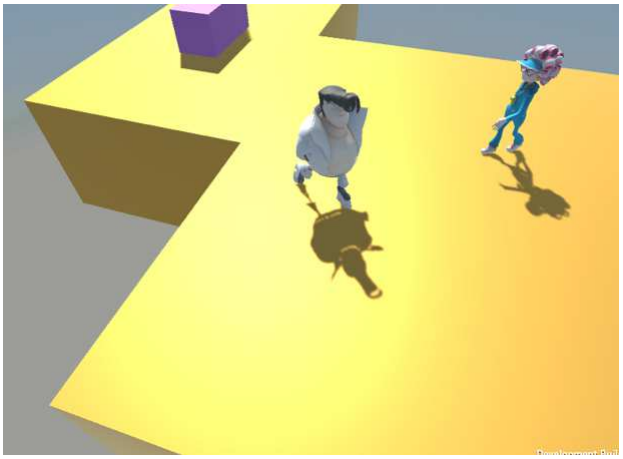


Figure 1: Screen shot from the game.

Elvis: The player character is a humanoid character, Elvis, who navigates about the scene using a *click-to-move* approach. The player clicks the mouse over desired point on the ground platform, and Elvis moves to this point. (This is done through a combination of Unity's navigation mesh and raycasting. Note that I explicitly want you to implement this form of navigation, as opposed to WASD or arrow-based inputs.)

You can use any humanoid model you want in place of Elvis, but your model should support at least two animations, an idle animation and a moving animation. Our player model is an Elvis impersonator called "Big Vegas" and was downloaded from the Mixamo Store, a repository that contains a number of character models and animations. I also downloaded from there two simple animations, an in-place jogging animation and an idle animation. (These will be made available to you if you want to use them. See the Projects section of the class web page.)

Grannies: Elvis is being chased by a number of non-player characters, called *grannies*. The grannies are also humanoid characters. They spawn at regular intervals. (In our implementation, there are three grannies. The first one spawns 3 seconds into the game and the other two spawn at 5-second intervals after this. Your are allowed to modify these values, but make the number small enough that the grader can easily win your game.)

The grannies start in an idle state. Whenever Elvis comes within a given distance threshold of a granny (we used 5 units), the granny starts chasing him. (If Elvis manages to escape beyond this distance, the granny returns to the idle state.)

If any granny catches up and collides with Elvis, then the game is lost. While each granny moves fast enough to make the game challenging (in our implementation the granny and Elvis have the same maximum speeds), grannies turn very slowly. The player wins by navigating near the edge of the platform and then stepping to the side at the last second, sending the slow-turning granny over the edge of the platform.

The granny character should support at least three animations: an idle animation, a moving animation, and a falling animation. Even though the grannies should not be able to the obstacles, you probably do not want your grannies using the navigation mesh, since they must have the ability to walk over the edge off. Their movement can be handled either explicitly by scripts or through the Unity physics engine. Raycasting can be use to determine whether granny has stepped off the platform.

Our granny model is called "Sporty Granny" and was also downloaded from from the Mixamo store along with her animations.

Camera Control: The camera should move with Elvis. In our implementation, we chose to have the camera facing Elvis diagonally, since this way you can see his face. (The Holistic3d tutorial cited below describes a simple method for setting up this sort of camera placement within the Unity editor.) You may place your camera wherever you like, but it should move with the player character.

If implemented correctly, the above elements are worth 80% of the grade. For full credit, the following additional elements should also be implemented.

Entry/Exit Screens: (5%) Rather than simply terminating the game, an informative message should be displayed to the player indicating whether the game is won or lost. There should also be an introduction scene that explains the game's objectives and inputs. (The introduction screen is not in our basic implementation.)

Pause/Restart/Quit: (10%) It should be possible to pause the action (and all the animations), restart, or quit your game at any time. (Pause is not supported in our basic implementation.)

Skybox: (5%) Elvis's floating mansion looks rather unimpressive under the default Unity skybox. I downloaded a prettier skybox, by MentalDreamAssets from the Unity Asset Store (see Fig. 2). You can use ours, which will be made available, our obtain your own.

As always, you are welcome to experiment with your own models, animations, and game semantics. The main elements (navigation meshes, animations and blending, click-to-move) must be present in some form. Do not worry too much about minor flaws in motion and

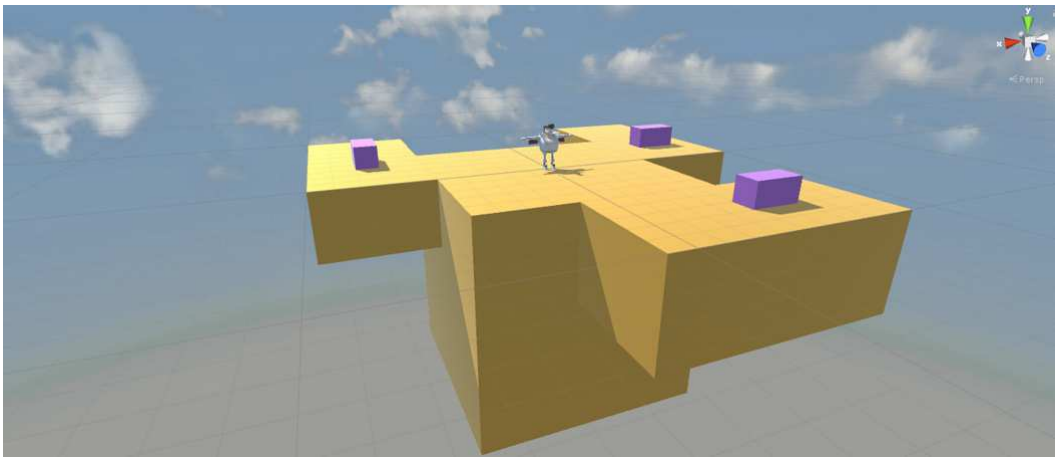


Figure 2: Elvis's golden floating mansion with enhanced skybox.

collision detection. Our main interest is that the animations are smooth, natural, and occur promptly when requested, and that the game is easy to play and test.

Online Resources and Tutorials: There are a number of online resources that were helpful in designing our implementation:

- https://unity3d.com/learn/tutorials/topics/navigation/navmesh-agent: The Unity video tutorial on NavMeshes. Also check out the videos under the section "Live Sessions on Navigation"
- https://docs.unity3d.com/Manual/Navigation.html: The Navigation section from the Unity manual. Check out the subsection on "Navigation How-Tos" especially "Moving an Agent to a Position Clicked by the Mouse" and "Coupling Animation and Navigation." (Note that with the simple animations that I used, I had some difficulty getting the method described in "Coupling Animation and Navigation" to work
- https://www.youtube.com/watch?v=*...rest of link omitted*: A tutorial by Holistic3d entitled "Mechanim & Mixamo in Unity 5". This explains how to create and download the Mixamo models and animations that we used in our implementation.

Our Implementation: We have implemented the programming assignment. The files can be found on the CMSC 425 Projects page. We will also create a file containing a lot of technical information about our implementation, which you may use if you like.

Final Submission: We will use the same procedure that we did for the first programming assignment (see the Submission Instructions from the Piazza post on Feb 5, 2017). Send an email to Alejandro with a link to a repository that contains a zip file with your submission. Also, remember to remove your Library and Temp directories, in order to keep the size small.

Programming Style: We will be reading your code to see that you implemented everything in a reasonable manner. Although programming style is not an explicit part of your grade, we reserve the right to deduct points for programs that are so poorly documented or organized that the TA cannot figure out how your program is working.

Optional Elements for Extra Credit: You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your Readme.txt file. The number of points of extra-credit credit will be left to the discretion of the TA.

External Resources: If you make use of any external resources in your program (or things that you developed prior to this class), even if you modified them, you *must* credit them in your Readme.txt file. Failing to do so will be considered an act of plagiarism. If you are unsure, check with me.

Homework 2

Handed out Thu, April 27 (and revised Mon, May 1). Due at the start of class Thu, May 4. Late homeworks will not be accepted (without prior approval), so turn in whatever you have done.

Problem 1. Short answer questions.

(a) In skinning, what is principal reason for binding a mesh vertex to multiple joints?
  (i) Allows joints to rotate in multiple directions (like a neck) as opposed to just one (like the middle joint of a finger)
  (ii) Enhanced efficiency when rendering very large meshes
  (iii) Allows the mesh to deform smoothly as the joints rotate
  (iv) All of the above

(b) In the Ramer-Douglas-Peucker algorithm for simplifying a polygonal curve, what is the criterion for selecting the next vertex to be added to the curve?

(c) How many dimensions are there in the configuration spaces for each of the following motion-planning problems. Justify your answer in each case by explaining what each coordinate of the space corresponds to.
  (i) Moving a line segment in 2-dimensional space, which may be translated and rotated.
  (ii) Moving a line segment in 3-dimensional space, which may be translated and rotated.
  (iii) Moving a pair of scissors in 3-dimensional space, which may be translated, rotated, and may swing open and closed.

(d) What is the principal advantage of selecting a path that travels along the *medial axis* of the free-space?

(e) Which of the following aspects of human sound perception allow us to determine the location that a sound comes from? (Select all that apply.)
  (i) Our auditory systems detect differences in the arrival times of sounds to each of our ears
  (ii) Sound waves reflect off our upper body, head, and outer ears in different ways depending on the origin of the sound
  (iii) People move their head purposively while listening to a sound to determine its origin

(f) In our description of the four elements of the boid model for flocking behavior (separation, alignment, avoidance, and cohesion), what is the purpose of *cohesion*, and how might it be implemented?

Problem 2. Consider the quadrilateral A and square B, shown in Fig. 1. The quadrilateral A's reference point is located at the origin, that is, $p_a = (0, 0)$, and the other vertices are at $(2, 0)$, $(1, 2)$, and $(0, 2)$. The square B's reference point is at $p_b = (3, 4)$ and its other vertices are at $(5, 3)$, $(6, 5)$, and $(4, 6)$. Object A can translate but not rotate.

The *configuration obstacle* of B with respect to A is the set of placements of A's reference point so that it overlaps B. Describe (draw clearly or explain with coordinates) the configuration obstacle of B with respect to A.

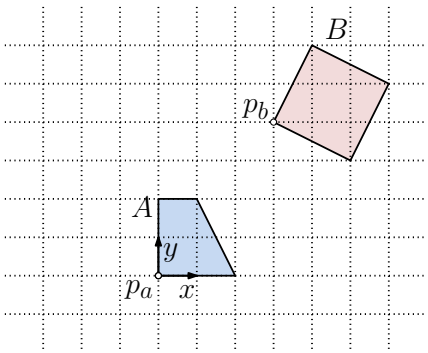


Figure 1: Problem 2.

Problem 3. In this problem we consider the performance of Dijkstra's algorithm and A* search on the graph shown below (see Fig. 2), where the objective is to compute the shortest path from s to t. Each edge (u, v) is undirected and is labeled with its associated weight $w(u, v)$.

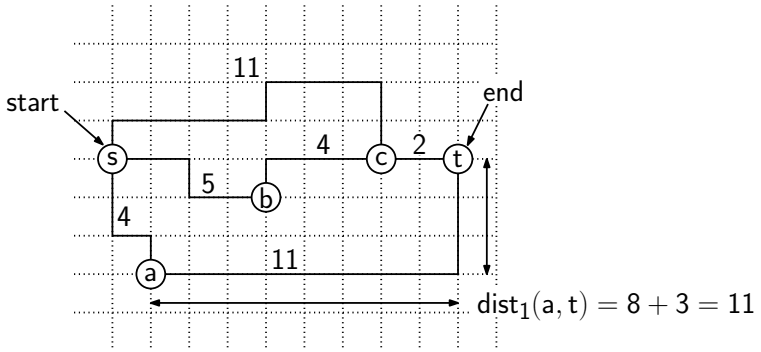


Figure 2: Problem 3.

For A* search we need to make use of a heuristic. To save you from dealing with square roots, as we did in the example from class, we will use the L_1 (also called Manhattan or checkerboard) distance between two points. It is defined to be the sum of the absolute values of the difference of the x and y coordinates of the points. For example, in the figure the L_1 distance between nodes a and t is $\text{dist}_1(a, t) = 8 + 3 = 11$. For A* search define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(a) = 11$. (For this graph it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

(a) Trace the execution of Dijkstra's algorithm on this graph. For each node indicate the following: (1) list the nodes in the order in which they are processed, (2) whenever a node is processed, indicate which of its neighbors have had their d-values modified, when the algorithm terminates (that is, when t is considered for processing), indicate what the final d-values are for all the nodes. If there are ties for which node is to be processed next, select the node that is earliest in alphabetical order. (As an example, see Lecture 16.)

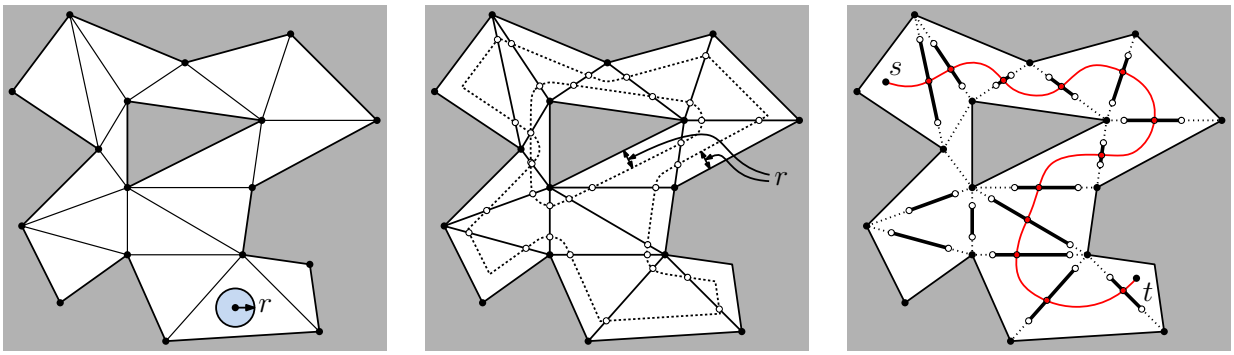
(b) Trace the execution of A* Search on this graph. Provide the same information as in

(a), but also provide the A* f-values for each node that is processed (recall that $f(u) = d[u] + h(u)$).

(c) Remark on the differences (if any) in the length of the path generated and the differences (if any) in the efficiency between these two algorithms.

Problem 4. One of the options that Unity (and other game engines) offer when dealing with Navigation Meshes is the ability to specify the size of the moving agent. Suppose that the moving agent is a circular disk of radius r (see Fig. 3(a)). One way to deal with this size is to "shrink" the accessible domain by this radius before triangulating the domain. In this problem, we will explore a different method. (This method has some additional advantages, since we can change the value of the radius without recomputing the navigation mesh.)

Here is the idea. Suppose that we have already computed a navigation mesh of the *entire* accessible domain (no shrinking). The mesh is represented by a triangulation T, which is stored using a DCEL data structure. The mesh contains two types of faces: the triangles of the navigation mesh, and obstacle faces. The obstacle faces are not required to be triangles. In order to identify what type of face you have, each face f of the DCEL has an additional field isObstacle. If f.isObstacle is true, then this face is an obstacle. Otherwise, it is part of the navigation mesh. (For example, in Fig. 3(a), the two shaded faces, the external face and the inner triangle, are both obstacles.) Throughout, we will assume that every vertex of the navigation mesh is adjacent to one obstacle face, and two obstacle edges.



(a) (b) (c)

Figure 3: Problem 4.

Given that our moving agent is a circular disk of radius r, to determine where its center point may navigate, we shrink the accessible domain by a distance of r. (This is shown in the dashed lines of Fig. 3(b).) This is called an *offset contour*. Rather than storing the entire offset contour, we will just store where it intersects the edges of the triangulation. (These are shown as white dots in Fig. 3(b).) Observe that the agent can safely move from one triangle to a neighboring triangle if the edge between these triangles contains a *window*, that is, a line segment that lies entirely within the portion of the mesh that is beyond the offset contour. (These window segments are shown as heavy lines in Fig. 3(c). Two edges of the triangulation in the figure are so short that they do not have a window, and therefore the agent cannot

travel across these edges.)

In this problem, we will investigate how to compute these windows and use them to compute paths in that navigation mesh that provide a clearance of r from the obstacle boundaries. **Note:** After writing the problem, I realized that there are conditions under which an edge may have multiple windows. To simplify matters, let's assume that this never happens. For example, I believe that if all the triangles of the navigation mesh are acute, each edge can have at most one window.

(a) Let e be any half edge of the navigation mesh such that e.left.isObstacle (see Fig. 4(a)). Using the operations of a DCEL, explain how to obtain the half edges e' and e'' that immediately precede and follow e, respectively, in counterclockwise order around the obstacle face. Also, explain how to obtain the vertices a and b at e's origin and destination, respectively. The operations should all take $O(1)$ time.

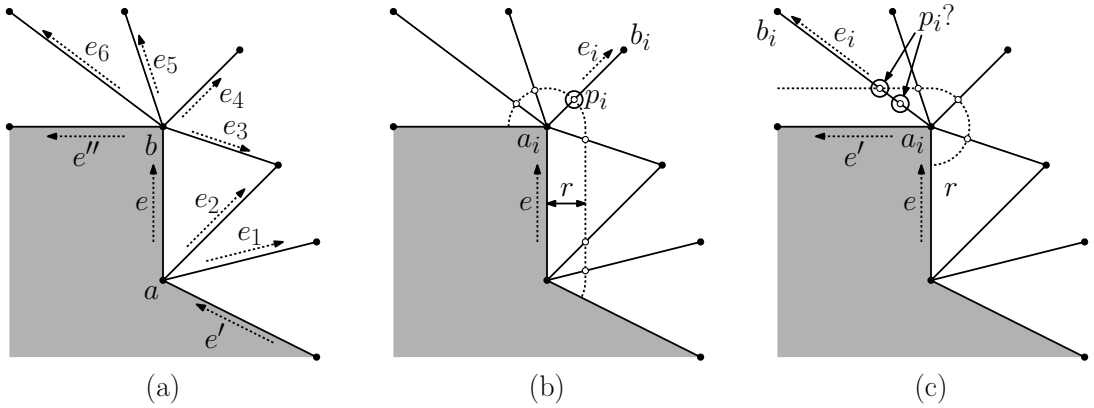


Figure 4: Problem 4 (continued).

(b) Using the operations of a DCEL, present a code fragment that returns a list L containing the half edges $\langle e_1, \ldots, e_k \rangle$ that are incident to a and b, not including edges that border the obstacle that is incident to e. (For example, in Fig. 4(a), L consists of $\langle e_1, \ldots, e_6 \rangle$.) These half edges should all be directed outwards from a and b, and the list should be computed in time proportional to the number of edges in the list.

(c) For any half edge $e_i \in L$, let a_i and b_i denote its origin and destination, respectively (see Fig. 4(b)). Given distance $r \geq 0$, explain (using the geometrical operations discussed in class) how to compute the point p_i along this edge that is at distance r from the obstacle edge e. (Consider only the edge e. The other incident edges e' and e'' will be considered in the next parts.)

Hint: I found it helpful for solving parts (d) and (e) to represent the point p_i as $a_i + \alpha \vec{v}_i$, where $\vec{v}_i$ is a vector aligned with the line segment $\overline{a_i b_i}$, but this is not a requirement. There are cases, depending on whether $a_i = a$ or $a_i = b$, and whether the angle between the (undirected) edges e_i and e is acute or obtuse.

(d) The point p_i computed in part (c) considers only one of the two obstacle edges incident on a_i. Assuming that we have applied part (c) to both of the edges incident to a_i, explain

how to compute which of these two points are to be used for the purposes of computing the window along this edge. (For example, in Fig. 4(c) we show two possible points p_i, one computed for edge e and the other for e'. How would you determine which of the two defines the endpoint of the window?)

(e) Given your answer to (d) and under our assumption that each edge has at most one window, for each triangulation edge (a_i, b_i), we have two window endpoints. One point p_i is at distance r from the obstacle incident to a_i, and the other point q_i is at distance r from the obstacle incident to b_i. Explain (using the geometrical operations discussed in class) how to determine whether the window segment for this edge is nonempty.

By the way, here is an explanation of how to complete the path computation, assuming you have successfully solved parts (a)–(d). Given a source point s and a destination point t, we first verify that both are at distance at least r from the nearest obstacle (otherwise there clearly is no path). Assuming so, we determine whether there is a path between them by computing the dual graph of the triangulation, where we keep only those dual edges whose corresponding windows are nonempty. Then, we determine whether the triangles containing s and t are connected by a path in this graph. Such a path passes from one triangle to the next through a window. It is not hard to show that if two edges of a triangle have nonempty windows, there exists a path of clearance r from one edge to the other. (You do not need to explain how to do this.)

Challenge Problem: In Problem 4, I observed that it is possible for a triangulation edge to have multiple windows. Draw an example to illustrate how this can happen. For a real challenge, prove that if all the triangles of the navigation mesh are acute, for any value of $r \geq 0$, each edge has at most one window.

Second Midterm Exam

This exam is closed-book and closed-notes. You may use two sheets of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (30 points, 3-8 points each) Short answer questions. Explanations are not required, but may be given for partial credit.

(a) The Unity mechanim animation system has a feature that can set the joint angles of a humanoid model to cause it to turn its head to face a particular direction. (Really!) This would best be described is an example which of the following animation techniques (select one):

(i) Keyframe animation

(ii) Motion capture

(iii) Inverse kinematics

(b) Our proposed algorithm for triangulating the walkable region in the construction of navigation meshes repeatedly cut off the *ear* such that the cutting edge has minimum length. What is an *ear* of a simple polygon? What is the reason for favoring short cutting edges?

(c) List one advantage and one disadvantage of using the *potential-field* approach to computing paths.

(d) In our description of the four elements of the boid model for flocking behavior (separation, alignment, avoidance, and cohesion), what is the purpose of *alignment*, and how might it be implemented?

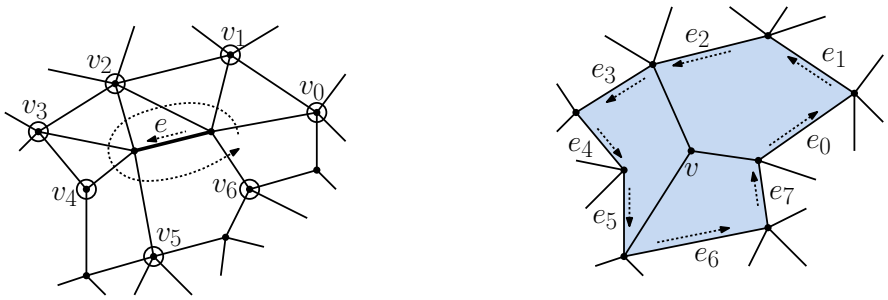
(e) Behavior trees have two types of task nodes, sequences and selectors. In a *sequence node*, its children are evaluated from left to right, and each returns either *success* or *failure*. Under what circumstances does the sequence node itself return success?

(f) Repeat (e), but this time for a *selector node*.

Problem 2. (25 points) Answer the following questions assuming that you are given a planar subdivision (i.e., cell complex of a 2-manifold) represented as a DCEL.

(a) Present a procedure (in pseudocode) that, given a half edge e of the DCEL, returns a list L consisting of the vertices that are adjacent to either of e's endpoints. The vertices should be listed in *counterclockwise* order about e. The list can start with *any* vertex, and *duplicates* are allowed.

For example, given the example shown in Fig. 1(a), the list $L = \langle v_0, v_1, \ldots, v_6 \rangle$ would be one valid result (as would any cyclical shift of this sequence). Your procedure should run in time proportional to the length of the output. (Hint: The answer is simpler if you choose the starting point carefully.)



(a) (b)

Figure 1: Problem 2.

(b) Given vertex v in a cell complex of a 2-manifold, the *link* of v is defined to be the edges that bound the faces that are incident to v, excluding the edges that are incident to v itself. (For example, in Fig. 1(b), the link of v consists of the edges $\langle e_0, e_1, \ldots, e_7 \rangle$.) Present a procedure (in pseudocode) that, given a vertex v of the DCEL, returns a list L consisting of the half edges of v's link. The half edges should be directed in *counterclockwise* order about v and the list should also be ordered in the same way. The list can start with any half edge of the link. Your procedure should run in time proportional to the length of the output.

Problem 3. (20 points) Consider the collection of shaded rectangular obstacles shown in the figure below, all contained within a large enclosing rectangle. Also, consider the triangular robot, whose reference point is located at a point s. (You may take s to be the origin.)

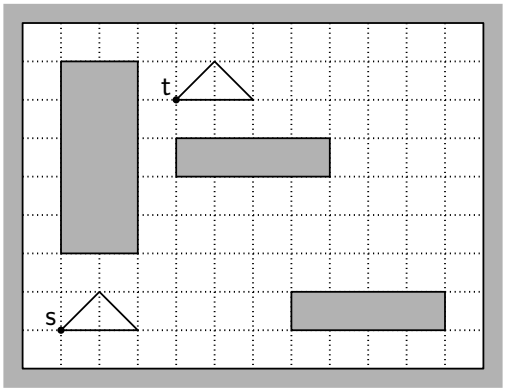


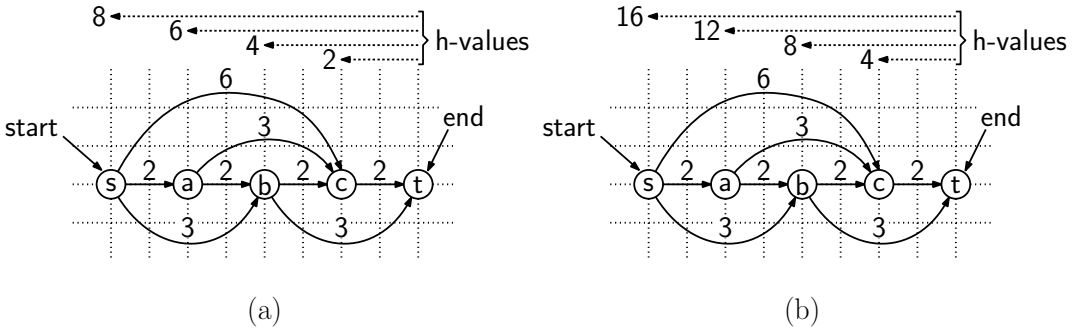
Figure 2: Problem 3.

(a) Draw the C-obstacles for the three rectangular obstacles, including the C-obstacle from region lying outside the large enclosing rectangle.

(b) Either draw an obstacle-avoiding path for the robot from s to t, or explain why it doesn't exist.

Problem 4. (25 points) In this problem, we will consider A* search under both admissible and inadmissible heuristics. *Please use the version of A* given in class.*

Consider the graph shown in Figure 3. For each node u, define $\text{dist}(u, t)$ to be the *straight-line distance* from u to t. For example, $\text{dist}(s, t) = 8$ and $\text{dist}(c, t) = 2$.



(a) (b)

Figure 3: Problem 4.

(a) Suppose that we take the *admissible heuristic* $h(u) = \text{dist}(u, t)$ (see Fig. 3(a)). (For example, $h(s) = 8$, $h(c) = 2$ and $h(t) = 0$.) Trace the execution of the A* algorithm on this graph using s as the start and t as the destination. In particular:

- list the nodes as they are processed, indicating the values of $d[u] + h(u)$
- whenever a node is processed, indicate how the d-values of its neighbors are updated

At the end (when t is processed), show the final d-values are for all the nodes.

(b) Suppose that we take the *inadmissible heuristic* $h(u) = 2 \cdot \text{dist}(u, t)$ (see Fig. 3(b)). (For example, $h(s) = 16$, $h(c) = 4$ and $h(t) = 0$.) Repeat (a) but using this different value of h.

(c) Did the algorithm produce the correct answer in part (b)? Explain briefly.

A* Search

```
A-Star(G, s, t) {
  foreach (node u) {                      // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered            // distance to source is 0
  repeat forever {                        // go until finding t
    let u be the discovered node that minimizes d[u] + h(u)
    if (u == t) return d[t]               // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))  // update d[v]
        mark v discovered
      }
      mark u finished                     // we're done with u
  }
}
```