Bro Dudley report

Hi class,

Once again, I'm providing email feedback for the milestone. These are reminders of things I'm generally grading for, plus maybe something you've done well or an optional suggestion for a way to make something better. I recommend attempting to apply these changes if possible since that will positively affect your project grade. The suggestions below are specific to your week 10 submission:

* Don’t forget to make FlyingObject’s destructor virtual.
* Don’t forget to make isAlive()/draw/getters const.
* Don’t forget to clean up your pointer memory in game.
* Lives, score and bonus don’t apply to all the children classes of FlyingObject, so they shouldn’t be stored in that class. Instead, put them in the classes where they’re used.

Good luck moving forward with the asteroids project

Brother Dudley

I'll just share my basic Asteroids class structure and a few thoughts, in case it helps anyone. I ended up renaming some things after looking at provided code.

FlyingObjects is an abstract base class. Mine has Point and Velocity plus game window extent data, and methods advance() and wraparound() plus pure virtual draw().

Bullet and Ship are both children of FlyingObjects that add particular traits. By the way, consider giving Ship a speed limit. Otherwise, if someone leans on the up arrow too long, the poor ship is wildly flicking across the screen, out of control (if it manages not to get hit).

Rock is another abstract base class, and also a child of FlyingObjects. All rocks have an angle attribute, used in overriden advance() and draw().

Then BigRock, MediumRock and SmallRock are all children of Rock. Each of them has a spawn() method, called when Game detects a collision. Spawn is passed a vector. Each type of rock knows what it breaks into when it is hit, and rocks.h has all the definitions in one place. Therefore, it builds the right smaller rocks with the right attributes and pushes them on the vector. After that, game is free to delete the broken rock, leaving a null in its place until cleanUpZombies() has a chance to tidy up the vector.

I haven't gotten far enough along to test spawn(), but see no reason it won't work. I had bullets but my method was clumsy. I'm redoing it to see if I can get more efficient with my code. Then I can add collision detection (bullet to rock and ship to rock) and have some fun.

There may be another way to handle the rocks breaking into other rocks. I'd love to hear anyone else's way of handling it.

Not spelled out in the game requirements is the idea that a ship has a fixed number of "lives" - 3 in the old Atari Asteroids. I may do that, and also declare loss if all 3 ship lives are used up, or victory if all the rocks are gone and the ship is still there.

I've never really understood sine / cosine, but with tinkering around and trial & error, I think I got it to work properly. It might be a bit fast, though, I'll have to check with the official build.  
  
Here's how I got it to work:

void Ship::applyThrust()  
{  
 velocity.setDx(velocity.getDx() + THRUST\_AMOUNT \* (-sin(M\_PI / 180.0 \* getAngle())));  
 velocity.setDy(velocity.getDy() + THRUST\_AMOUNT \* (cos(M\_PI / 180.0 \* getAngle())));  
}

[**David Hendricks**](https://byui.instructure.com/courses/51104/users/62427)

Yesterday Jul 4 at 4:14pm

[Manage Discussion Entry](https://byui.instructure.com/courses/51104/discussion_topics/763252?module_item_id=4188006)

I added addDx() and addDy() like those in Point, and it gets even simpler:

void Ship::thrust()

{

   Velocity v = getVelocity();

   v.addDx(THRUST\_AMOUNT \* (-sin(M\_PI / 180.0 \* angle)));

   v.addDy(THRUST\_AMOUNT \* (cos(M\_PI / 180.0 \* angle)));

   // I put a test for excessive speed here as a condition  - your choice

   setVelocity(v);

}