



ASTRONAUT VS ALIENS CODERS' GUIDE



Introduction

Project Goal: Create Your Own Game

The video and computer gaming industry is huge! It has sales of between 85 and 100 billion dollars and billions of players around the world. Clearly, it's fun playing computer games, but it's even better to create your own game. Once you've created a game, you'll have a better understanding of how your favourite games are built, why certain design choices were made, and what you might have done differently.

In the following six lessons, you are going to create a simple seek and pursuit game. An astronaut needs to escape a planet and return to earth. First, he / she needs to get 3 solar panels to power their space ship while escaping aliens and without running out of time or lives. Once you have the basics, you can change game conditions like lives or speeds, to see how these changes affect gameplay. You can also choose a different game theme if you do not like astronauts and aliens.

This project uses a development process in which you'll be repeatedly cycling through the process of planning, designing, implementing, and testing as you refine your project. This is a commonly used approach to software development called an *iterative* and *incremental* process.

Plan

Before you start, here are some important tips:

- In the All Projects menu, there is a folder called Games! Open it and play then look at the Astronaut Game sample project for ideas on how to start your project and see what is possible.
- It is important to remember to save your project often! Click on the **Arrow going up to the Cloud** icon.



Learning objectives - Here is what you'll do in this project:

- Plan your work. Break it into stages or smaller steps.
- Repeatedly cycle through your work as you implement new features, test, and debug,
- Use the Lynx commands to create turtle movements and animation.
- Add interactive objects, such as buttons and interactive turtles to trigger actions and restore starting position.
- Use multiple sprites to create animations

Introduction

- Use “random” events.
- Add event handlers to be used as triggers, such as collision and colour events.
- Add programming features to set up and run your simulation multiple times.
- Use conditional statements (if statements) with logic operators.
- Create and use different types of variables, including those associated with text boxes and sliders as well as global and local variables.
- Add music and/or sound to improve the game experience

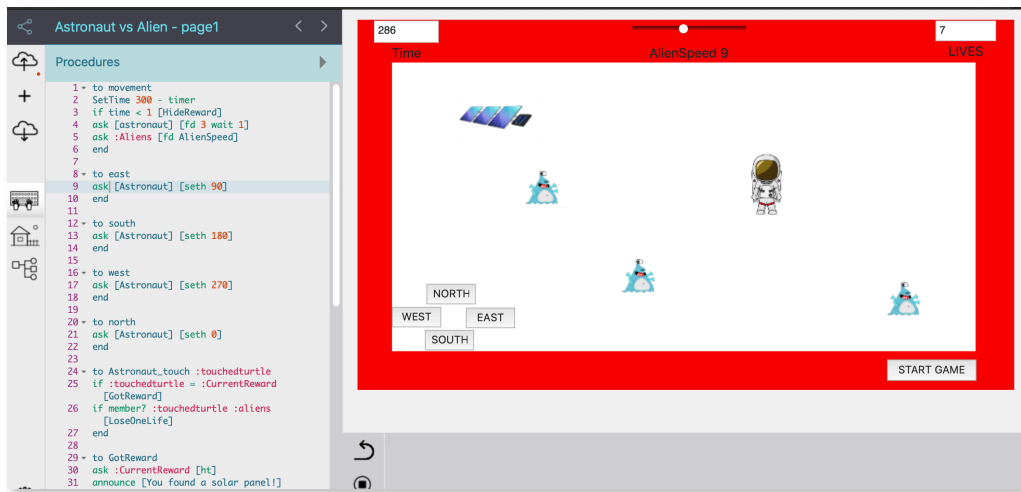
BEFORE YOU START: This project assumes you already have a Lynx account and are familiar with the various components of the Lynx screen. *We also assume you have made other Lynx projects before starting this project.*

Lesson 1 - Start With a Plan

Step 1: Explore the Astronaut Game sample

Click on **All Projects** on the Lynx main page, open the **Games** folder, and select the **Astronaut vs Alien** project. The project appears in **Play Mode**. Start the Game! To see how it was created and also to make changes, log in to Lynx and click the green **Edit** button for this project.

The project opens in the Lynx Editor.



While playing with the project, think about what you already know how to do and what's new. Consider what you like about the project and what you would probably do differently.

Step 2: Plan your work

Although it's exciting to jump in and begin a new project, it's important to start by first focusing on your game's functional design. There are several things you should think about *before* you start coding.

1. **Know your goal!** The end project looks (and works!) better when you start with a plan and you know what your goal is.
2. Decide on some project parameters. How many rewards (solar panels) will the Astronaut have to find? Will the rewards be solar panels or something else? How many lives will the astronaut have? Will it be an Alien pursuing the Astronaut or something else? In our sample, an Astronaut is trying to find three solar panels to attach to the space ship but is also trying to elude three aliens, but you should use your imagination.

Lesson 1 - Start With a Plan

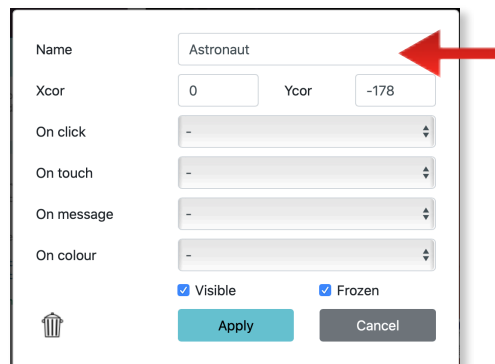
3. **Create a first draft!** Start by creating a first draft of the project that includes the main project components. As a coder, it is important to plan and create your code before you spend time on the visual design features (Tip: *Choose your final clipart last.*) Go over the project several times - each time adjusting and fine-tuning it to achieve your original goals. This is called an **iterative** approach to development.
4. **Save time to polish your project!** Remember to save some time at the end for polishing the final project by, for example, adding special clipart and sounds or music.

Step 3: Start Your Game

After you have logged-in at <https://lynxcoding.club/> click on **Create a Lynx Project** on your **My Projects** page.

As with all new projects, there's a turtle in the Work Area. The turtle will be your main player. Since there will be other characters in this game, it's important to give the turtle a meaningful name, for example, **Astronaut**.

Right-Click on the turtle and in the **Name** field delete **t1** and type **Astronaut**, then click **Apply**.



Leave the Astronaut set to the turtle shape so you can always see the direction it is heading.

Start by drawing a game field. This is the area in which the game will be played. It is like a boundary. Use the Astronaut turtle to draw wide bars at the top and bottom of the screen and at the two sides.

First drag the turtle to the top left-hand corner of the Work Area.



Lesson 1 - Start With a Plan

Here's How!

How do you figure out how long to make the bar? There are a few ways:

- Trial and error – this works, but can take some time.
- Remember the screen size you selected for your project – Standard size is 800 x 450.
- Check the position of the turtle. In the Command Centre, type:

```
show xcor  
-395
```

You may get a slightly different number

This shows the distance from the turtle's position on the left edge, to the horizontal center of the work area (since you're looking at distance, you can disregard the negative sign) and so it is approximately half the screen width.

Using any of the above methods to calculate the length, draw a wide bar by following these instructions. Type in the Command Centre:

```
setheading 90  
pendown  
setpensize 50  
setc 'red'  
fd 790
```

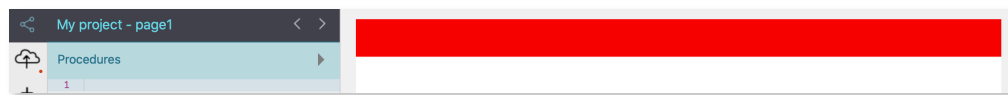
or `seth.`

or `pd.`

Sets the line thickness.

Replace red with blue, green.

Use the width you calculated.



NOTE: It may be difficult to see your Astronaut turtle now (**red on red!**) so type `setc 'black'` to change the Astronaut's colour back to black. Don't forget the quotation marks around the colour name.

There is a Lynx Colour Chart in the **User Guides** section of the **Help** tab at lynxcoding.club.

Move the turtle to the bottom, left-hand corner of the Work Area and draw a bar at the bottom of the screen.

Next, using the same process as described above, create thick bars at the sides of the Work Area. You can use `show ycor` to determine the **forward** distance required.

Your Work Area should look like this:

Lesson 1 - Start With a Plan



Step 4: Controlling the Astronaut

Next, let's control the Astronaut.

In the Procedures Pane, define a movement procedure for the Astronaut:

```
to movement
ask [astronaut] [fd 3 wait 1]
end
```

As you work, you may want to adjust your inputs for `fd` and `wait`

Although there is only one turtle that is following instructions right now, this project will include a number of different turtles. `ask` temporarily asks the named turtle to follow a command without changing the current turtle, the turtle that has been following (listening to) commands. It's important to always be aware of which turtle will react to commands.

Test the procedure in the Command Centre.

```
pu
setc 'black'
forever [movement]
```

Remember to pick the pen up first!
Choose any colour you want

`Forever` runs the *Movement* instruction in the square brackets forever or until it's stopped either with a `stopall` command or by clicking the square **Stop All** icon just to the left of the Command Centre.



Now that the Astronaut is moving, create a navigation system that uses buttons to change the Astronaut's direction.

Lesson 1 - Start With a Plan

Start by writing direction procedures for the buttons in the Procedures Pane, for example:

```
to east
```

```
ask [Astronaut] [seth 90]
```

```
end
```

Make sure you have brackets around `[seth 90]`.

```
to south
```

```
ask [Astronaut] [seth 180]
```

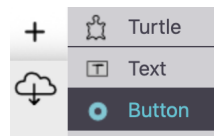
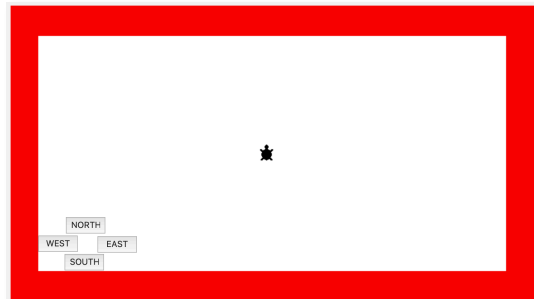
```
end
```

Remember to **ask** the Astronaut to follow the instructions, just as you did in your **movement** procedure.

Think about the heading for **West** and **North** and add these two procedures.

Next, create four buttons that will run these four procedures.

Right click on each button and give each one an appropriate label for example, **EAST**. In the **On click** event handler, select the corresponding procedure name for that label. Click **Apply** when you are done and do this for all four headings.

A screenshot of the Scratch 'Button' configuration dialog box. The 'Name' field is 'button1'. The 'Label' field is 'EAST'. The 'On click' dropdown menu is set to 'east'. There are checkboxes for 'Visible' (checked) and 'Frozen' (unchecked). At the bottom, there are 'Apply' and 'Cancel' buttons.

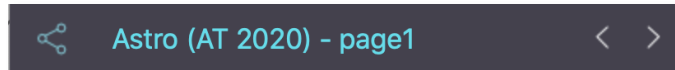
Test your buttons with the Astronaut. It will not move, but it should change its heading as you click on buttons.

Remember to use semicolons (;) to add comments *in, before or after* a procedure in the Procedures Pane to clearly identify what it does. This will *really help* your friends understand the logic and purpose of your procedures. Commenting is a great habit of professional programmers.

Lesson 1 - Start With a Plan

Step 5: Save!

If you haven't yet done so, give your project a name by clicking on the project name above the Procedures Pane and typing in a **new name**.



Then SAVE!

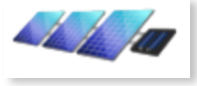
Lynx does not automatically save your work, so save your projects often!

Lesson 2 - Adding Solar Panel Rewards!

Step 1: Add Solar Panel Rewards

The Astronaut needs to be seeking something. In the sample game, the Astronaut is trying to get three **solar panels** (but, of course, you can use whatever clipart or shape for your reward that you want). Start by creating the first reward.

1. Find clipart you want to use for a reward. Choose an empty cell in the Clipart Pane and click on the "+" sign in the bottom right corner. Remember the number of this clipart.
2. Add a turtle and remember its name (`t1` for now). You do not need to change this turtle's name. You'll see why shortly.
3. Set this turtle to the clipart you chose as a reward image. In the Command Centre, type:
`t1, setsh 1` (or whatever number your reward shape is).



TIP: You can always use the `setsize` command to adjust the size of the reward turtle / solar panel. The default `setsize` is 40.

4. Place this solar panel turtle somewhere in the Work Area.

Step 2: Capturing the Reward

Use an **On touch** event to let the Astronaut know it has reached the solar panel reward.

Right-click on the Astronaut, click on the down arrow next to **On touch** and select **New**. Then click **Apply**.

Name	Astronaut
Xcor	7
On click	-
On touch	New...

Now check the Procedures Pane.

A new procedure appears in the Procedures Pane called `Astronaut_touch`. You need to add instructions to this new procedure. What do you want to have happen once a solar panel reward is touched (obtained)?

In the following `Astronaut_touch` procedure, if the Astronaut touches the reward turtle, it

Lesson 2 - Adding Solar Panel Rewards!

runs another procedure (which you will define shortly).

```
to astronaut_touch :touchedturtle
if :touchedturtle = 't1' [GotReward]
end
```

The `if` instruction sets up a conditional statement: If something is **true**, do whatever is in the square brackets. If it is **false**, do nothing and go to the next instruction (if there is one).

Although you could just list some responses in the square brackets, it's a good idea to use a subprocedure (a procedure which is called by another procedure) since you may want to add responses to the **On touch** event as you add features. Using subprocedures makes it easier to edit these instructions and easier to read and understand your Procedures Pane.

The procedure `GotReward` isn't defined yet. Write a new procedure in the Procedures Pane:

```
to GotReward
end
```

What do you want to have happen when the reward is obtained (touched)? Edit your `GotReward` procedure, for example, to have the reward disappear:

```
to GotReward
ask 't1' [ht]                ht stands for Hide Turtle
announce [You found the solar panel!]
end
```

Test your game components.

Remember, if you ever need to show your hidden turtle, type this in the Command Centre:

```
t1, st or ask 't1' [st]
```

Step 3: Adding More Solar Panel Rewards

Add two more turtles: type this in the Command Centre:

```
repeat 2 [clone 't1']
```

Place the three solar panels in three distant spots in the gaming area. When the Astronaut finds a reward (when he /she touches a solar panel), you will have to know and do the following things:

- Know which solar panel reward should hide (which one was touched);
- Show the next reward;
- Know when the last reward is found.

Lesson 2 - Adding Solar Panel Rewards!

You could write separate programs for each Solar Panel reward turtle, but a more efficient way of handling these actions is to use *variables*, in this case two global variables. The value for the first variable is a list of all the unfound rewards and the value for the second is the current reward.

Since the variable for the list of all unfound Solar Panel rewards needs to be *reset each time* a new game is started, put it in a procedure that *sets up* or *initializes* your game, for example:

```
to initgame
make 'AllRewards [t1 t2 t3]
ask :AllRewards [ht]
end
```

Creates a variable named 'AllRewards' and sets its value to [t1 t2 t3]. Use your turtles' names.
All turtles in the variable :AllRewards hide.

The `InitGame` procedure sets up the initial complete list of rewards, in this example, three rewards. This procedure will be used to initialize other conditions as your game develops.

To test this procedure, type in the Command Centre:

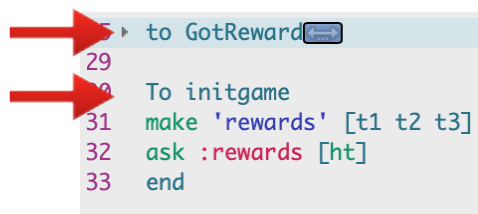
```
initgame
show :AllRewards
```

show the value of the variable :AllRewards

You should see:

```
t1 t2 t3
```

Note: You can "collapse" your procedures in the Procedures Pane by clicking on the triangle beside `to` in any procedure. This way you can see the overall structure easily and your Procedure Pane isn't so cluttered! This does **NOT** work if `to` begins with a capital `T`.



Lesson 2 - Adding Solar Panel Rewards!

Step 4: Create a Second Variable

Next you need to know which specific reward the Astronaut is seeking in each round. Remember, this changes as the game is being played. You'll also need a way to automatically change the list of unfound solar panel rewards each time one of the rewards is found by the Astronaut.

The following procedure carries out these actions:

```
to NextReward
if empty? :AllRewards [winner]
make 'CurrentReward first :AllRewards
make 'AllRewards butfirst :AllRewards
ask :CurrentReward [st]
end
```

First, examine the `NextReward` procedure line by line:

```
if empty? :AllRewards [winner]
```

`Empty?` reports `true` if its input (a word or list) is empty. If it is, Lynx runs the instruction in the brackets, in this case, `winner` a subprocedure that is not yet defined. `:AllRewards` will only be empty if the Astronaut has found all the Solar Panel rewards. This line checks if that is true.

As you go through the procedure, you'll see how `:AllRewards` arrives at an empty state. This instruction line is the first line of this procedure to avoid encountering a bug in the next line.

```
make 'CurrentReward first :AllRewards
```

This line creates the second global variable defined as the `first` element in `:AllRewards`. So, if the value of `:AllRewards` is `[t1 t2 t3]`, the value of `:CurrentReward` would be `t1`. The variable `:CurrentReward` represents only one reward.

```
make 'AllRewards butfirst :AllRewards
```

Here, `make` resets the value of `:AllRewards`. It's now everything it was before *except* (*but*) the *first* element. So if `:AllRewards` had been `[t1 t2 t3]`, then

`butfirst :AllRewards` is `[t2 t3]` (`bf` is the short form of `butfirst`).

```
ask :CurrentReward [st]
```

The turtle identified by the variable `:CurrentReward` shows.

Lesson 2 - Adding Solar Panel Rewards!

Add the `NextReward` procedure to the Procedures Pane and then create a `winner` procedure to celebrate winning the game, for example:

```
to winner
announce 'You win! Hooray!'
stopall           Stops all processes.
end
```

You may want to add other features to this procedure later.

Try the procedure a few times and, as you do so, check the values of both the `:AllRewards` and `:CurrentReward` variables. First stop the moving Astronaut. In the Command Centre, type:

```
stopall
```

Then type:

<code>initgame</code>	This hides the rewards.
<code>nextreward</code>	This shows one of the rewards.
<code>show :AllRewards</code>	You should see t2 t3
<code>nextreward</code>	This shows the next reward
<code>show :AllRewards</code>	You should see t3

Run these two lines again and again. At one point, the list should be empty, and you will see the Winner message in the **Announce** box.



Don't forget - Save your project!

Lesson 3 – Enemy Aliens

Step 1: Using the Variables

Before adding any enemies to chase the Astronaut, use the variables you created to make your procedures work for all the Solar Panel rewards.

In **Lesson 2, Step 2**, you created an `astronaut_touch` procedure and a `GotReward` procedure for `t1` only, but now you have three rewards. This is where the variables you created to represent the specific reward being touched comes in handy.

Edit the `astronaut_touch` and `GotReward` procedures to make them work for *each* of the Reward turtles at the appropriate time.

```
to astronaut_touch :touchedturtle
if :touchedturtle = :CurrentReward [GotReward]
end

to GotReward
ask :CurrentReward [ht]
announce [You found the reward!]
NextReward
end
```

This keeps the game going.

Test your procedures. First run the `InitGame` procedure, then `NextReward` then type `forever [movement]`. Although you're still missing parts of your game, at this point, just make sure these work as you expect.

It's a good idea to create a `StartGame` procedure to run these procedures:

```
to startgame
initgame
nextreward
forever [movement]
end
```

Add a **Start Game** button in the Work Area to run this procedure.

Step 2: Adding Enemies

You have the bare bones of a game, but a game is more fun when there are some challenges. In the sample game, the Astronaut is chased by its enemies, which in this sample are Aliens, but you can use any enemy shape.

Find the shape you want to use for your enemies, add it to the Clipart Pane, and take note of the shape number.

Lesson 3 – Enemy Aliens

Add three turtles to the work area.

Set the Alien turtles to the shape you've chosen.

Type in the Command Centre:

```
ask [t4 t5 t6] [setshape 20]
```

Use your shape's number and turtles' names. Use `setsize` to adjust their size if need be.

Since you'll be frequently addressing these turtles as a group, it's a good idea to create a variable that stands for all of the Aliens.

Edit your `InitGame` procedure to create a new variable:

```
to InitGame
make 'AllRewards [t1 t2 t3]
ask :AllRewards [ht]
make 'aliens [t4 t5 t6]
ask :aliens [setheading random 360]
end
```

Add this line to create a variable. Use your turtles' names.

Each alien's heads in a different direction.

You don't want to hide any of the Alien turtles!

Step 3: Chasing Movements

Next, you need to get your Aliens to move so they can try to bump into the Astronaut. Since the Aliens need to keep moving *as the Astronaut moves*, the instruction can be added to the `movement` procedure.

Add a new instruction in the `movement` procedure:

```
to movement
ask [astronaut] [fd 3 wait 1]
ask :aliens [fd 5]
end
```



Lesson 3 – Enemy Aliens

Step 4: Debugging

Test all your procedures: just click on the **Start Game** button, then on your your **N, S, E, W** buttons.

Is everyone moving? When you click a direction button is the right turtle following the instructions?

As a project gets more complex, bugs begin to pop up. *All programmers get bugs* in their programs – that's why companies have quality assurance programmers and beta versions of their software!

Take the time to make sure you understand why a bug appeared and what you can do to prevent it from happening. Debugging is an important process and shouldn't be left to the end.

Make sure you've used the correct syntax, for example, check that you didn't use a colon (:) where you need a quotation marks (' ') or vice versa. Review all the steps in this project so far to make sure you clearly understand when to use each one.



Don't forget to Save often!

Lesson 4 – Events

Step 1: Astronaut Meets Alien!

Although the Astronaut and Aliens are all moving, nothing is happening when they meet. What should happen once they meet? Here are some ideas:

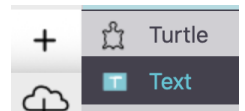
- The Astronaut disappears, there's a loud noise, and the game is over.
- The Astronaut hides and then pops up in a new random place to start again.
- The Astronaut loses either a life or a point or runs out of time.

All of these options are possible. In the first two, the game is either over too quickly or lasts forever. The third option lets you set up some parameters to keep track of the Astronaut's encounters.

Give the Astronaut Lives!

Start by giving the Astronaut Lives and a way to lose or gain them. Although you could create a global `lives` variable, you can also represent these values by using a text box. Text boxes not only display the number of lives available, but help calculate any changes to the number of lives. They are like variables that you can actually watch!

Create a text box by selecting **Text** in the “+” menu.



Right-click on the text box and give it a name that clearly shows what it is tracking, for example, **Lives**. Remember, a text box name must be one word with no spaces. The box will only contain a number so it doesn't need to be large.

Type a number in the text box, for example, 5. Leave the name of the box showing.



When a text box is created, Lynx generates **two new words**. One is the **name** of the text box, which is used to **report** the text box's contents to an appropriate command. For example, if you type in the Command Centre:

```
show lives
```

you get this response

```
5           Or whatever number you typed in the text box.
```

The second word generated lets you **set** the text box's contents or value. In the Command Centre, type:

```
SetLives 15
```

The number in your text box should change to **15**.

Now reset the **Lives** text box to **5**. This is the number of lives an Astronaut has left, at the beginning of a game.

Lesson 4 – Events

When should the Astronaut lose a life?

One time an Astronaut would lose a life should be when it touches an Alien. You already have the `Astronaut_touch` procedure that contains instructions for when an Astronaut touches a **Reward**. Edit that procedure to include instructions for what to do when the Astronaut touches an **Alien**.

```
to astronaut_touch :touchedturtle
if :touchedturtle = :reward [gotreward]
if member? :touchedturtle :aliens [LoseOneLife]
end
```

Member? checks if the turtle being touched is one of the elements of the variable `:aliens`. If it is, it runs `LoseOneLife`, a new procedure that will be defined net.

What should happen when `LoseOneLife` runs? The Astronaut should lose a life. Define a `LoseOneLife` procedure.

```
to LoseOneLife
if Lives = 0 [stopall]
SetLives Lives - 1    This sets the new value of Lives to whatever it was at first minus 1
end
```

Test your procedures. Run the game, and let collisions happen until the text box reaches zero.

Step 2: Astronaut Gets Solar Panel Reward

Astronauts should also be able to gain lives if they capture a Solar Panel Reward. How can you make this happen?

You can increase the value of Lives as well as decrease it. Edit your `GotReward` procedure to give the Astronaut an extra life when it finds a Solar Panel Reward.

```
to GotReward
ask :CurrentReward [ht]
announce [You found a solar panel!]
setLives Lives + 1
NextReward
end
```

Lesson 4 – Events

Edit your `InitGame` procedure to set up the initial value of the Lives text box.

```
to InitGame
make 'AllRewards' [t1 t2 t3]
ask :AllRewards [ht]
make 'aliens' [t4 t5 t6]
ask :aliens [setheading random 360]
```

SetLives 5 Use the starting value you want, you can change your mind later.
`end`

Check your edited procedure. Play your game a few times to make sure everything works. Take a little time to review all your procedures to make sure you understand what each one does.

REMEMBER: you can "collapse" your procedures in the Procedures Pane by clicking on the triangle beside `to` in any procedure.

Step 3: Changing Speeds

To add another challenge to your game, create a slider that will let you change the Aliens' speed.

Select **Slider** in the "+" menu. Right-click on the slider and give the slider a meaningful, one-word name, such as **AlienSpeed**. Again, use a single word, no space.



Just as with text boxes, when creating a slider, Lynx automatically generates two new words, one that **reports the current value** of the slider, the other that lets you **set the value** by way of commands.

Since this slider will be used to set the speed of the Aliens, you want to limit the slider's range. Right-click on the slider again and set an appropriate maximum and minimum level, for example, a maximum of 20 and a minimum of 5.

Name	AlienSpeed
Min	5
Max	20
Value	5

Lesson 4 – Events

Next, edit the `movement` procedure so that the Aliens move forward the speed selected on the slider.

```
to movement
ask [astronaut] [fd 3 wait 1]
ask :aliens [fd alienspeed]      AlienSpeed reports its value to fd.
end
```

Also, edit your `InitGame` procedure to set the initial value of the **AlienSpeed** slider.

```
to InitGame
make 'AllRewards [t1 t2 t3]
ask :AllRewards [ht]
make 'Aliens [t4 t5 t6]
ask :Aliens [setheading random 360]
setAlienSpeed 5      Choose whatever number you want within the range you set.
end
```

To increase the game challenge, get the Aliens' speed to automatically increase each time the Astronaut captures a reward. For example,

```
to GotReward
ask :CurrentReward [ht]
announce [You found a solar panel!]
SetLives Lives + 1
SetAlienSpeed AlienSpeed + 2
NextReward
end
```

You've probably noticed that you need to go back and edit your procedures as you add features and improve your game. This iterative and incremental process is normal in the development of video games and many other types of programs. This process lets you carefully move from a simple design to a more complex one, debugging as you go.



Test, test, test! And Save, save, save!

Can you also add a slider to change the Astronaut's speed, a **AstronautSpeed** slider?

What procedure do you edit to incorporate this and how, where, and when can you increase the speed? Remember to initialize the speed in the `InitGame` procedure.

Lesson 4 – Events

Step 4: Reviewing and Refining

Take some time now to review your game to determine what is missing. For example, currently when the Astronaut runs out of lives, the game ends, but no additional information appears. What else could happen once the game is over?

In the `LoseOneLife` procedure, when available Lives equal zero, everything stops (`stopall`). Instead of just stopping everything, you can either add some commands here to describe other reactions or you can create a `GameOver` subprocedure that is general enough to use when there are no more lives or, if you add more features, when other conditions are met. This second method gives you more flexibility.

```
to GameOver :reason          :Reason is a local variable that is used only in this procedure.
announce word 'Game over! ' :reason
stopall                      Everything stops when a game is over.
end
```

`word` combines its inputs into one word and reports this word to another command, in this case `announce`.

Next, edit your `LoseOneLife` procedure.

```
to LoseOneLife
if Lives = 0 [GameOver 'You ran out of lives.']
setLives Lives - 1
end
```

`'You ran out of lives'` replaces the local variable `:reason` in the `GameOver` procedure. You need a single quotation mark at the start and end of the phrase.

An input on the title line of a procedure is a local variable. A local variable holds its value only while Lynx is running **that** procedure. That means that when running the procedure, the procedure requires an input (like `forward` or any other built-in command that requires an input).



Remember to save!

Lesson 5 – Raising the Game Level

Step 1: Adding Boundaries

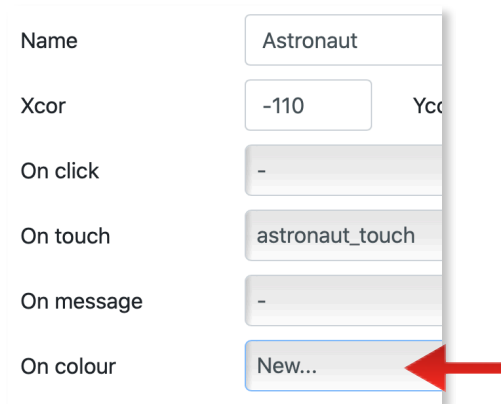
You now have all the main pieces of your game. It's time to consider what other features would enhance game play.

You may have noticed that once the Astronaut and Alien reach the coloured bars at the edges of the Work Area, they cross them and seem to leave the game field, reappearing at the opposite end of the field. It's as if they've gone around the back of the screen or wrapped around it like a ribbon if you were tying a package.

To keep all the elements within the game field at all times, make the bars true boundaries. Get the Astronaut or the Aliens to react when they touch a colour.

Start by creating an **On colour** event handler for the Astronaut.

First, right-click on the Astronaut, click on the down arrow next to **On colour** and select **New**. Click **Apply**.



A new procedure appears in the Procedures Pane.

```
70 ▾ to Astronaut_oncolour :prevColour :newColour
71   ; Use an instruction like this to do colour
    detection every time
72   ; Moving from any colour to red, even red to
    red, will trigger the action.
73   ; Pick your own colour name and instructions
    instead of [BACK 10 RIGHT 180]
74   ; IF :NEWCOLOUR = "RED [BACK 10 RIGHT 180]
75   end
```

Read the comments and delete them. Note that there are two local variables in this procedure, `:prevColour` and `:newColour`. As the Astronaut moves, it checks the colour under it and gauges a change in colour (from a previous colour to a new colour).

Lesson 5 – Raising the Game Level

Consider this program:

```
to Astronaut_oncolour :prevColour :newColour
if equal? :prevColour :newColour [stop]
if :newColour = 'red' [back 7 right random 360]
end
```

Look at it line by line:

```
if equal? :prevColour :newColour [stop]
```

This means if the previous colour is the same as the new colour, do the instructions in the square brackets, which in this case is to stop the procedure. If not, Lynx goes to the next line.

Note this uses `stop`, not `stopall`. `stop` only stops the procedure that it is part of `stopall` does what it says – stops everything.

```
if :newColour = 'red' [back 7 right random 360]
```

If new colour is a specific colour, in this case **red**, the instructions in the brackets run.

Remember to use the colour that you used for your boundaries.

In the instructions for the `if` statement, the Astronaut backs up a little and turns a random amount. Using `random` means the Astronaut's movements are less predictable.

It's good to note that without the first instruction line in this procedure, every time the Astronaut moves over **red**, even if it had been on red before, it would follow the instruction for **red**. The first line reacts only to a *change* in colour.

Create a similar procedure for the Aliens. Since this procedure will be used by all your aliens, create an `alien_oncolour` procedure in the Procedures Pane first.

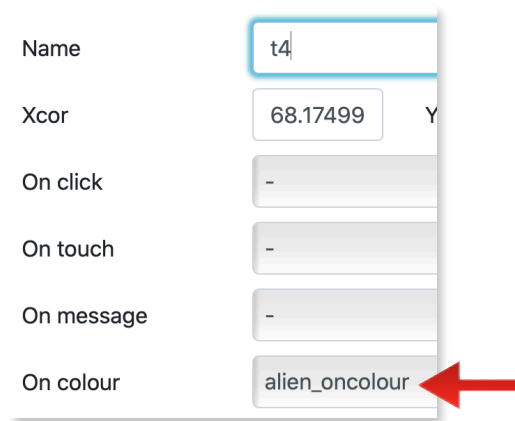
Then select this procedure in each Alien's **On colour** event handler.

For example:

```
to alien_oncolour :prevColour :newColour
if equal? :prevColour :newColour [stop]
if :newColour = 'red' [bk 10 random 360]
end
```

Lesson 5 – Raising the Game Level

Next, it is important to open the dialog box of **each** Alien and select this procedure in the **On colour** event handler.



The screenshot shows a dialog box for an Alien object. It has several fields: 'Name' with the value 't4', 'Xcor' with the value '68.17499', and 'Y' with a partially visible value. Below these are event handler fields: 'On click', 'On touch', 'On message', and 'On colour'. The 'On colour' field is highlighted with a red arrow pointing to it, and its value is 'alien_oncolour'.

Step 2: Upping the Game – Exploring the Timer (Optional)

Instead of letting the Astronaut have an unlimited amount of time to reach each Solar Panel reward, set a time limit.

Lynx has a built-in timer that can be used to create a calibrated way to keep track of elapsed time. This timer runs automatically and is not controlled by procedures. As you'll see, the only way to control this timer is to reset it to 0.

First, create a text box that will keep track of the time available. Give it a meaningful name, such as **Time**.



Set the value of the **Time** text box by typing in the Command Centre:

```
SetTime 300
```

Next, type this in the Command Centre:

```
show timer
```

```
342
```

You will get a different number.

Timer shows time elapsed since the last time the timer was reset or since Lynx was started.

Type in the Command Centre:

```
resett
```

```
show timer
```

```
21
```

You will get a different number.

Lesson 5 – Raising the Game Level

To keep track of elapsed time, you can

- set the **Time** text box to a specific value (300 for example) at the start and then each time a Reward is found,
- reset the built-in `timer` at the same time (so back to zero),
- then, each time the Astronaut moves, note the `timer` value at that instant and subtracting it from the value in the **Time** text box.

Edit your `movement` procedure to include a way to keep track of time elapsed.

NOTE: To test the next changes, you must

1. add the new instructions to the `movement` procedure,
2. add a `HideReward` procedure, and
3. edit your `InitGame` procedure.

```
to movement
```

```
SetTime 300 - timer
```

```
if time < 1 [HideReward]
```

```
ask [Astronaut] [fd 3 wait 1]
```

```
ask :aliens [fd AlienSpeed]
```

```
end
```

This subtracts elapsed time as measured on the Timer from the original value of Time

`HideReward` is a new procedure not yet defined.

Why not use `if time = 0 [HideReward]`? Sometimes the value of **Time** may end up being less than 0. Using `time < 1` will report **true** when both the value of `time = 0` and when the value is less than 0.

The value of `time` is recalculated every time `movement` runs.

Note: Did you previously add a AstronautSpeed slider? If so, your movement procedure would look like this.

```
to movement
```

```
settime 300 - timer
```

```
if time < 1 [HideReward]
```

```
ask [Astronaut] [fd AstronautSpeed wait 1]
```

```
ask :aliens [fd AlienSpeed]
```

```
end
```

Lesson 5 – Raising the Game Level

Next, what happens when time runs out? Put these instructions in the `HideReward` procedure. For example:

```
to HideReward
ask :CurrentReward [ht]
GameOver 'You ran out of time!'
end
```

Use your `GameOver` procedure again.

Finally, you need to have a way to set the **Time** text box and reset the **Timer** to **zero** each time a new reward appears for the Astronaut to find. The logical place to add these instructions is to the `NextReward` procedure.

```
to NextReward
if empty? :AllRewards [winner]
make 'CurrentReward first :AllRewards
make 'AllRewards butfirst :AllRewards
SetTime 300
resett
ask :CurrentReward [st]
end
```

Why not also edit the `InitGame` procedure?

You could add these instructions to the `InitGame` procedure, but you also would need to add them to the `NextReward` procedure. This is because the Timer and the Time text box are reset each time a reward is found. Since the `NextReward` procedure runs immediately after the `InitGame` procedure when you start the game, you need only include the instructions one time in the `NextReward` procedure.

Now, test your procedures!



Remember to save your project!

Lesson 6 – Final Steps

Step 1: Find a clipart for your Astronaut

Just like you did for the reward and the alien, go find a clipart for the Astronaut shape. Using the turtle original shape was good for observing the direction and the functionality of your game while you were working on it. Now is the time to go for the looks!

Click on the "+" in an empty clipart spot, note the clipart number, and use a `setshape` instruction to give that shape to the Astronaut turtle. WAIT!

Before running the `setshape` instruction, make sure the RIGHT turtle is listening to the instruction. You can either

- click on the Astronaut turtle before running the `setshape` instruction, or
- designate the turtle by its name using `ask`:

```
ask 'Astronaut' [setshape 4]
```

or using the comma method:

```
Astronaut, setshape 4
```

Step 2: Adding Sound Effects

Adding sound effects makes a game more fun. Where can you add them? How can you improve the game-feel by adding them? Consider adding a sound when the Astronaut finds a reward, when time runs out, or even when the game starts. Are there other times to add sounds?

Step 3: Designing the User Interface

You have all the pieces for your game. How does it look? Do you need to tweak the layout? A well laid out UI is part of good game design.

For example, it's easiest for the user if the direction buttons are placed in the following pattern:



Where is your button to start the game? Is it in an easily seen place so players know when to start? Buttons should be arranged so that a player can understand how to start and play just by looking at the game.

Lesson 6 – Final Steps

You should add a text box, **announce** alert or front page for your project that contains game instructions. Create a button that runs a procedure that either shows the announce alert, shows and then, after a few seconds or hides an instruction text box or moves from the instruction page to the game.

Once you have decided where to put the buttons, freeze them in place.

```
freeze [button1 button2 button3 button4 button5]
```

You may also want to freeze the **astronaut**, the **rewards**, and the **aliens**. They'll still follow instructions, but a player won't be able to cheat by dragging them!

Step 4: Using the Arrow Keys (Optional)

To add another option to the game, use the arrow keys on your keyboard instead of buttons to change the Astronaut's direction.

There are several commands you can use to read and report whatever keyboard characters are pressed. Every key on the keyboard has a corresponding ASCII code (ASCII stands for American Standard Code for Information Interchange). To determine an ASCII value, type in the Command Centre:

```
show ascii 'A'      Capital A.  
65
```

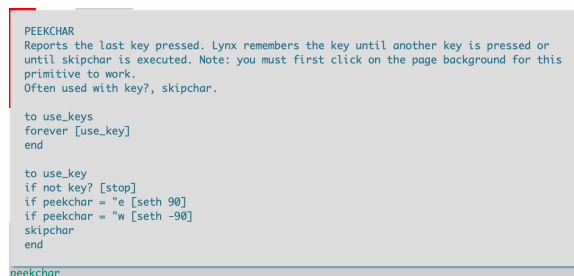
It's more difficult to use this method to find the ASCII values for keys other than letters and numbers, but there are charts available online.

Once you know an ASCII value, you can use it to determine what the key is or to call that key:

```
show char 65  
A
```

By using these codes, you can create actions triggered by various key presses. Look at the tooltips following additional primitives to better understand how you can program various keys:

peekchar, **key?**, **skipchar**.



Lesson 6 – Final Steps

Here is the beginning of a procedure to use:

```
to direct
if not key? [stop]           If there's not a key pressed, stop only this procedure.
if peekchar = char 28       If the key pressed is the left arrow, whose ASCII code is 28...
[ask [Astronaut] [seth 270]] ...the Astronaut sets its direction West.
skipchar                   Forgets the last key pressed so the memory is cleared for
                           the next time the procedure runs.
end
```

You'll also need to define a procedure that keeps running the `direct` procedure, for example:

```
to UseKeys
skipchar                   This ensures the memory is cleared of past key presses.
forever [direct]
end
```

Test these procedures before you add the other directions.

Add the additional lines for the other keys. The ASCII codes for the other directions are:

- Right-arrow = 29
- Down-arrow = 31
- Up-arrow = 30

Can you add the appropriate lines to the `direct` procedure? Continue the following procedure.

```
to direct
if not key? [stop]
if peekchar = char 28 [ask [Astronaut] [seth 270]]
                           Remember to have two ending brackets!
if peekchar = char 29 [ask[astronaut][seth 90]]
...
...                         The code for the other two arrows...
skipchar
end
```

NOTE: `Skipchar` is added just once, at the end of the procedure.

Test your procedures!

`UseKeys` should run when the Astronaut and the Aliens are set in motion, so add it to `startgame`.

Test your game. If you use the keyboard to drive the astronaut, you may want to declutter your game field by unfreezing and deleting the direction buttons.

Lesson 6 – Final Steps

IMPORTANT: After collecting a solar panel (and clicking **OK** in the **announce** box), you need to click again in the Work Area to re-enable your arrow keys. You **HAVE** to click in the Work Area for **peekchar** to work.

Step 5: Share your Project.

When you have finalized and saved your game and are ready to go public, there are several ways to share your project. You should talk to your teacher about the best way to share your projects.

FROM THE LYNX EDITOR

- From within the Lynx editor, simply click on the **Share** icon in the top-left corner of the editor.
- In the dialog box that comes up, select **Project Properties** and choose an image file to use as a preview. (Use the **Preview Image Select** button to get a file from your device). Enter a title and a description.
- Then click on **Sharing Options** and click on **Create** a link to share. Copy the link to paste it where it is needed *or* click on **E-Mail** to send the link by email.
- You can also share via **Twitter** or **Facebook**.

FROM WITHIN YOUR LYNX PERSONAL SPACE IN THE CLOUD

- From your Lynx personal space, click on your project to open it in **Play Mode**.
- Then click on **Share** and then follow the steps described immediately above.

ENJOY (AND EDIT) MY PROJECT

- You can not only let your someone play your game, but also let them edit it.
- Before sharing a project, go to its **Properties** in your Lynx personal space in the cloud.
- Uncheck the **Private** check box.
- Click on the **Share** button and then the **+** sign in the **URL** field and **Copy** the Link.

Send the link to a friend. Using the link, he / she will be able *to make and save changes to the game* **but your original game remains the same**.