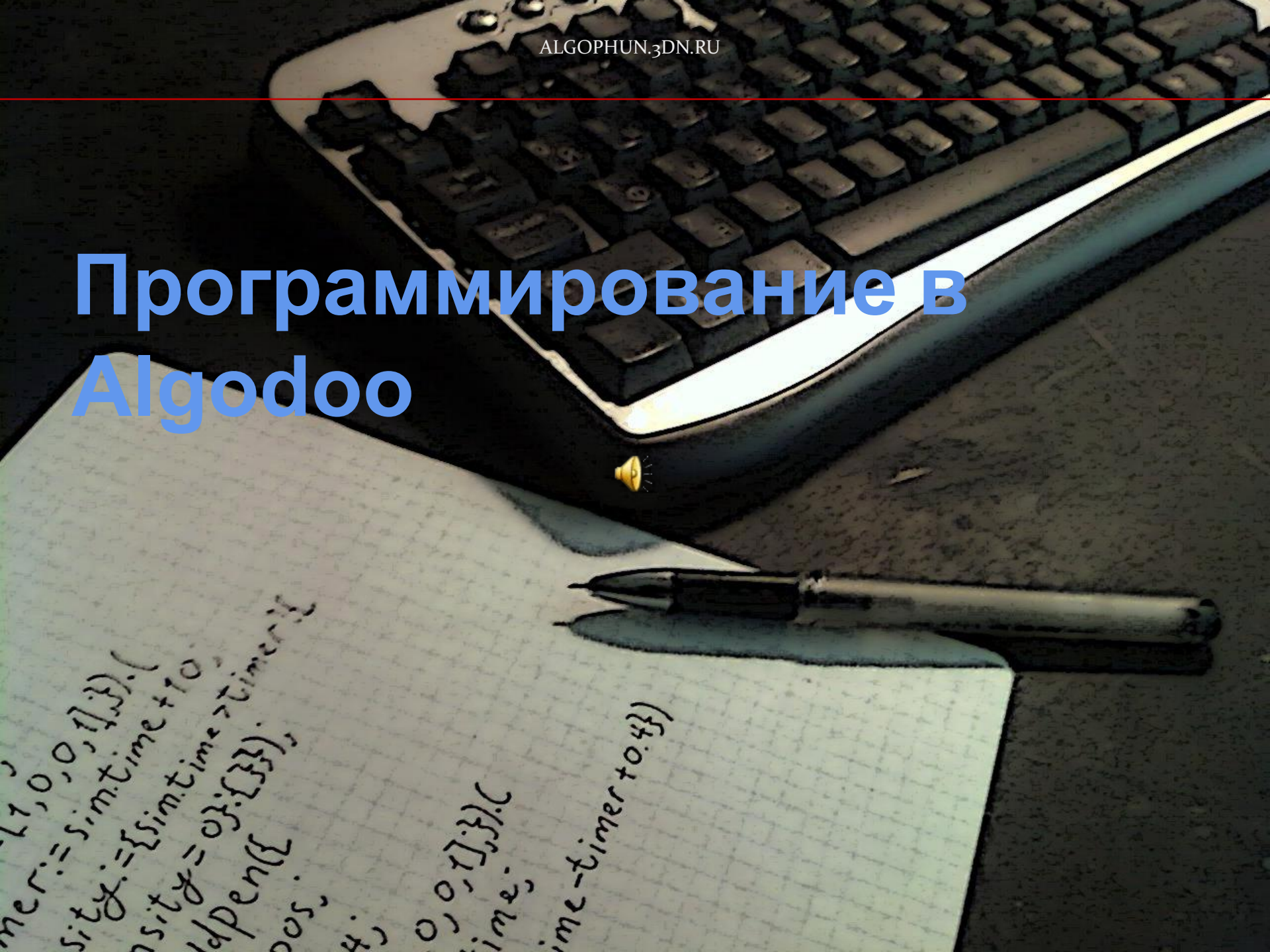
Программирование в Algodoo



Программирование в Algodoo.

Издание второе, дополненное.

Владивосток
ALGOPHUN.3DN.RU

2011

Содержание.

Содержание.

От автора:

На сегодняшний день особенно остро встал вопрос об уровне знаний среди пользователей Алгоду, особенно ярко это проявляется среди пользователей рунета. В то время когда англо-говорящие пользователи заходят на официальный сайт и там получают помощь, то наши соотечественники не имеют такой возможности из-за простого незнания языка. Поэтому я поставил себе цель сделать все для повышения «квалификации» русских пользователей. Эта книга – попытка систематизировать информацию о программировании в Алгоду, а как хорошо она получилась – судить вам. С радостью приму ваши отзывы и предложения на нашем сайте – Algophun
На сегодняшний день особенно остро встал вопрос об уровне знаний среди пользователей Алгоду, особенно ярко это проявляется среди пользователей рунета. В то время когда англо-говорящие пользователи заходят на официальный сайт и там получают помощь, то наши соотечественники не имеют такой возможности из-за простого незнания языка. Поэтому я поставил себе цель сделать все для повышения «квалификации» русских пользователей. Эта книга – попытка систематизировать информацию о программировании в Алгоду, а как хорошо она получилась – судить вам. С радостью приму ваши отзывы и предложения на нашем сайте – Algophun.3
На сегодняшний день особенно остро встал вопрос об уровне знаний среди пользователей Алгоду, особенно ярко это проявляется среди пользователей рунета. В то время когда англо-говорящие пользователи заходят на официальный сайт и там

Введение

Привет! В этой книге я рассажу и так называемых скриптах и их использовании. Вы наверняка слышали о них, видели в действии и хотели бы научиться их использовать в своих сценах, но никак не могли найти хоть сколько-нибудь понятного урока (я знаю о них, некоторые сам писал, теперь даже стыдно взглянуть на это, так сказать, творчество). Если вы не слышали и не знаете, не беда – сейчас расскажу ☺

Как известно, программы для компьютеров написаны на специальных языках программирования. Что же они из себя представляют? Набор правил, по которым происходит чтение текста, и которые определяют, что значит тот или иной набор символов и что делать том или ином случае. Все это нужно чтобы правильно обработать информацию. То есть работа компьютера или программы заключается в следующем

Ввод информации
Хранение информации
Обработка информации
Вывод информации

Чтобы было понятнее, приведу пример из повседневной жизни. Помните школьные контрольные? Особенно по какому-нибудь тяжелому и противному предмету вроде алгебры? Вспомнили? Хорошо, теперь проследим процесс с точки зрения программиста:

Вы идете к приятелю с вопросом «как решать эти примеры?» – ввод информации.

Чтобы не забыть это, вы написали шпаргалку – теперь в ней хранится информация.

На контрольной вы прочитали шпаргалку, и стали думать, как решить ваш пример по этому принципу – обработка информации.

Наконец решив задачу, вы записали решение в тетрадь и сдали ее учительнице – вывод информации.

Примерно так же работают и программы. Но, давайте вернемся к нашему Алгоду. В этой игре есть встроенный язык программирования, называемый **Thyme**. С его помощью вы можете обрабатывать информацию, полученную из игры и использовать для различных целей. Как пример привожу сцены, в которых с помощью скриптов сделаны различные автоматы, роботы, оружие, просто различные спецэффекты и прочее, и прочее…

Так, скажете вы, что это за скрипты такие? Слово «скрипт», также как и «код» обозначает отрывок или весь текст, записанный по правилам данного языка программирования. Понятно, что все что вы там понапридумываете и запишете, будет являться скриптами, поэтому выражение «Ого, это наверно все скрипты из игры», с которым я встречался, в корне неверно.

Урок 1: Синтаксис в Thyme

Давайте посмотрим, из чего состоит Thyme.
Если проанализировать какой-нибудь код, то можно условно разделить использованный там Thyme на несколько операций – операция объявления, операция присваивания, условная операция и вызов команд/функций. Рассмотрим каждую из них.

Операция объявления:
  идентификатор := значение
Идентификатор это имя переменной/функции/команды (вы узнаете, что это такое), а значение это информация которую она будет хранить. После выполнения этой операции в памяти программы будет создана новая переменная/функция/команда с указанным значением.

Операция присваивания:
  идентификатор = значение
В этом случае будет изменено значение уже существующей переменной, если же она не существует, могут возникнуть непредвиденные ошибки.

Условная операция:

условие ? действие1 : действие2

При выполнении этой операции будет проверяться, выполняется ли условие и если да, то будет выполнено действие1, в если условие не выполняется – действие2.

Вызов функции:

функция(аргумент1, аргумент2)

Вызов команды:

команда

Подробнее про функции и команды вы узнаете позже.

При перечислении нескольких операций они разделяются точкой с запятой

идентификатор = значение ; функция(аргумент1, аргумент2)

Также следует помнить о том, что создавать переменную можно только один раз, если попытаться создать переменную с уже существующим именем, то вы увидите сообщение об ошибке, но уже существующей переменной будет присвоено новое значение.

Возможно, сейчас вы не совсем поняли про синтаксис, но я сейчас расскажу про переменные, и все станет понятно.

Урок 2: Переменные, функции, команды и прочие страшные слова

Вы уже знаете, что программе требуется где-то хранить информацию. Так вот, переменная и является хранилищем информации. То есть после того как мы объявили новую переменную программа записала в своей памяти что переменная имеет имя такое-то и значение такое-то. Вопросы о том, что это за память, как производиться эта запись и пр. не столь важны, поэтому не стоит задумываться о них, а следует запомнить, что через имя переменной мы можем получить информацию, которая в ней хранится.
Насчет памяти. В принципе, количество переменных не ограничено, не думаю, что вы сможете создать сколько-то там тысяч (или миллионов, я не вдавался в подробности) переменных, которые займут всю свободную память…. Поэтому можно не боясь создавать любое нужное количество переменных.

Значение переменной это не просто кусок текста с потолка, оно может быть одного из пяти типов:
Integer – целые числа, такие как 1, 2, 5, -2, 0 и проч. Хочется сказать про отрицательные числа – они записываются в круглых скобках.

Float – дробные числа, такие как 0.3, 0.7, 1.0, 3.6 и проч.
String – текст, пишется в двойных кавычках, например "просто текст", но сами кавычки не учитываются.
Boolean – особый тип, у него есть только два значения - True (истина) и False (ложь). Он используется во многих случаях, когда нужно не значение, а своеобразный «выключатель».
Array – массив, также возможно название «список» (List). Как ни очевидно, но это список из нескольких значений. Значения заключены в квадратные скобки и разделены запятыми, например [1, True, 0.3, 0.5, "Привет, мир!"]

Ну вот, вместо скучной теории началась практика («Ура!!!»). Давайте запустим наш любимый Алгоду и приступим. «Эмм, как же мне работать с переменными?» - очень просто, нажмите клавишу "F11" (если ничего не произошло, нажмите "~"), и вашему взору предстанет **Консоль**. В верхней части консоли отображаются предыдущие записи и сообщения, а внизу строка, в которую вы можете записать свой код, нажать Enter, и он выполнится.

Давайте попробуем, введите в консоль код наподобие такого

```
var := 13
```

И нажмите Ввод. Если вы увидите знак > и ваш код, а внизу значение 13, то значит что все хорошо.

Поздравляю, вы только что создали свою первую переменную!
Если вспомнить что я писал, то видно, что var это имя переменной (идентификатор) (вы еще часто его встретите, не удивляйтесь, это просто сокращение от англ. Variable - переменная), а 13 это значение. Thyme знает простые математические операции, они обозначаются +, -, /, * . Можете убедиться сами, введите в консоль

```
var*1.5
```

И, о чудо, вы получите ответ – 19.5.

Проведем эксперимент. Нажмите "Вверх" два раза, вы увидите, что можно пролистывать историю ваших записей, и подтвердите код

```
var := 13
```

«Погодите-ка, ведь мы уже объявили эту переменную?» - да, все так и есть, и вы получите подтверждение в виде ошибки сообщающей что «Данная переменная уже существует».

Теперь можете поэкспериментировать еще, например, создайте еще переменных, присвойте одной из них значение другой…. Хочу только заметить, что имя переменной должно состоять из латинских букв и не должно содержать пробелы, вместо них можете использовать _

Наигрались? Теперь давайте посмотрим, зачем нам переменные.
Создайте круг, прикрепите к нему один конец пружины, а второй фону. В меню «Пружины…» посмотрите длину пружины. Теперь присвойте одной из ваших переменных значение с длиной пружины (вбейте число, нажимая по клавиатуре ☺).
Сделали? А теперь, самое главное!
Выделите пружину, откройте контекстное меню. Почти в самом конце списка увидите вложенное меню «Скрипты». (Теперь-то вы знаете, что это за скрипты такие ☺).
Вы увидите некоторое количество строк, в левой части которых записано свойство, а в правой – поле для ввода (Вам это ничего не напоминает?).
Найдите строку Length, и введите такой код

{var}

Нажмите Enter. Теперь запускаем симулятор… на первый взгляд все как обычно. Но только на первый, зайдите в консоль и измените значение вашей переменной. Хм, что-то изменилось.… Так, надо проверить… Ага, вот оно! Теперь вы можете контролировать длину пружины с помощью скриптов. В этом весь смысл, что с помощью скриптов вы сможете сделать так, чтобы свойства нужных объектов автоматически изменялись, чтобы создавались новые объекты, изменялись параметры работы программы.… Если вы видели сцены со скриптами, то вы представляете себе это великолепие.

Вы уже догадались, что таким образом можно контролировать не только длину пружины, но и любое свойство. Давайте расскажу, как это происходит.
Если вы обратили внимание, то наверно заметили, что строки в меню скриптов очень напоминают переменные – идентификатор есть, значение есть. Именно так оно и есть, эти переменные со свойствами записаны в информации об объекте. Точнее, информация об объекте это и есть набор переменных описывающих его свойства….
И когда мы ввели var в строку Length, то сделали пойнтер – зависимость одной переменной от другой. А то что эта зависимость не пропала сразу после того как мы нажали Enter и Length получила новое значение, объясняется тем что мы ввели фигурные скобки. Можете проверить, прописывая код со скобками или без них.

Урок 3: Структуры

Если вы не против, давайте порассуждаем о переменных. Если против, можете просто почитать ☺

Итак, переменные используются для работы программы. Их там вагон и маленькая тележка (если не верите, можете проверить, сейчас расскажу как). Поэтому в голову сразу приходит мысль, что хранить навалом такое огромное количество переменных не есть хорошо и надо бы их как-то систематизировать.

Разработчики тоже не дураки и именно так и сделали. Вся память делится на разделы, разделы на подразделы, которые содержат переменные….

Как пример – раздел App, в котором лежит подраздел Background который содержит переменные, относящиеся к Фону. Например, за цвет фона отвечает переменная

App.Background.Skycolor

По умолчанию ее значение равно [0.45, 0.55, 1, 1], но вы можете его легко изменить.

Увидеть список переменных очень легко. Откройте консоль, нажмите Tab. Вы увидите список разделов и неструктурированных переменных. Если вы введете одну или несколько букв и нажмете Tab, то консоль выведет список переменных/разделов, которые начинаются на эту комбинацию. Если таких нет, то ничего не произойдет, а если раздел всего один, то она допишет его до конца и выделит заглавные буквы (конечно, нет абсолютно никакой разницы между написанием заглавными или строчными буквами, но когда за неимением пробелов слова пишут вряд, это помогает).

Все это хорошо, но вы можете использовать только уже существующие разделы.

Вот вроде бы и все про структуры, пошли дальше.

Урок 4: Scene.my.*

Давайте представим небольшую ситуацию. Вы сделали свою первую сцену со скриптами (ничего особенного, но для вас и это много), сохранили ее, выключили компьютер, пошли погулять, а когда вернулись и открыли сцену.… Обругали разработчиков, меня, программу и компьютер не очень хорошими словами.
Что же случилось? Просто вы неожиданно обнаружили, что ваши скрипты не сохранились, а переменные бесследно исчезли.… Не унывайте, еще не все потеряно. Просто все нестандартные переменные, функции и команды не сохраняются. За исключением тех переменных, что вы создали в разделе Scene.My.
Так что просто запомните, что объявлять переменную стоит таким образом -

```
Scene.my.newVar := 13
```

Урок 5: Массивы

Когда вы читали о типах значений переменных, то запомнили что есть такой тип – массив, и что у него есть несколько значений. Действительно, это так, поэтому массивы удобно использовать для хранения большого (или не очень) количества однородной информации, например, цвет объекта хранится как массив из четырех чисел -

```
color = [1, 0.3, 0, 1]
```

Где элементы значат [Красный, Зеленый, Синий, Непрозрачность]

Для того чтобы извлечь значение из массива, следует поставить после его имени пару круглых скобок с номером элемента внутри (элементы нумеруются целыми числами начиная от нуля). Пример -

```
density = color(0)
```

Массивы могут иметь очень большое количество элементов, например, массив с вершинами полигона –

```
surfaces = [[[0.1990676, 0.15117884], [-0.04093218, 0.231179], [-0.1009326, 0.25117898], [-0.16093206, 0.27117896], [0.39906693, -0.09882188], [0.31906748, 0.031178474], [0.27906752, 0.071178675], [0.23906755, 0.11117864]]]
```

Но иногда они не очень удобны, так как нельзя изменить отдельный элемент. То есть, такой код -

Урок 6: Команды и функции

Вам наверняка хочется узнать, что такое команды и функции, которые я упоминал в начале книги, поэтому можете обрадоваться – момент настал.

Что же такое команда? Это связка из идентификатора и действия. Действием может быть любой скрипт... Если вы опять не совсем поняли, то привожу пример –

```
scene.my.command := {scene.my.var = scene.my.var * 2 + 1}
```

Теперь, вместо того чтобы каждый раз прописывать это действие (scene.my.var = scene.my.var * 2 + 1), вы можете просто прописать название команды. Попробуйте сами, вы увидите, что если это действие надо выполнить много раз в разных случаях, гораздо удобней и быстрей писать только название команды.

Отлично, тогда что такое функция? Функция это такая разновидность команды, для которой надо указывать один или несколько аргументов.
Пишутся функции тоже очень легко –

```
scene.my.Function := (X)=>{scene.my.var = x + 3}
```

В круглых скобках указываются аргументы, которые надо вводить, потом вы можете использовать внутри самой функции.

Но это еще не все, существуют команды и функции, которые возвращают значения.
Например, такой код -

```
scene.my.com := {(scene.my.input*10+3)/10};
scene.my.var = scene.my.com
scene.my.var2 = scene.my.com
scene.my.var3 = scene.my.com
```

равнозначен такому

```
scene.my.var = (scene.my.input*10+3)/10
scene.my.var2 = (scene.my.input*10+3)/10
scene.my.var3 = (scene.my.input*10+3)/10
```

Аналогично дело обстоит и с функциями, вот пример, вместо такого кода -

```
scene.my.var1 = math.mod(math.cos(scene.my.in1*2+5), 12)
scene.my.var2 = math.mod(math.cos(scene.my.in2*2+5), 12)
```

можно написать

```
scene.my.fun := (i)=>{math.mod(math.cos(i*2+5), 12)}
scene.my.var1 = scene.my.fun(scene.my.in1)
scene.my.var2 = scene.my.fun(scene.my.in2)
```

Главное здесь – не бояться скриптов…

Урок 7: Массив функций и команд

Массив может содержать в себе не только обычные значения, но и функции с командами. Так же как и обычные массивы, они служат для систематизации информации, только тогда – значений, а сейчас – действий.

Пишется такой массив также как и обычный, только внутри лежат действия команд или функций.

Пример –

```
scene.my.ActionArray := [(x)=>{x*10+3}, {scene.my.var=13}]
```

Используется он тоже просто –

```
scene.my.var = scene.my.ActionArray(0)(7)
```

если действие от функции, а в случае если от команды –

```
scene.my.ActionArray(2)(0)
```

Урок 8: Условные операции

В начале книги я упоминал условные операции, теперь настало время рассмотреть их поближе. Структура “Если” это инфиксная операция, позволяющая задать вопрос, и определить действие выполняемое если условие выполняется или не выполняется.

Есть два способа написать условную операцию – инфиксный и стандартный.
Для стандартного используется функция if_then_else –

if_then_else(x, y, z)

Где X – условие, Y – действие на истину, Z – действие на ложь.
Инфиксный способ короче и привычнее –

X ? {Y} : {Z}

Для написания условий используйте следующие знаки
==, =!, <, >, <=, >= - думаю, понятно, что они значат.
Например –

```
scene.my.var>10?{scene.my.var=0}:{}
scene.my.bool == true ?{}:{scene.my.bool = true}
scene.my.bool?{}:{scene.my.var = 3}
```

Но не всегда после условия пишутся действия, тогда скобки можно не писать –

```
density = sim.time>13 ? 0 : 2
```

Урок 9: onCollide

Давайте немного отвлечемся и поговорим о других, простых, но не менее важных вещах ☺.

OnCollide, onHitByLaser, onLaserHit – все это функции, которые выполняются при столкновении двух тел, тела и лазера или лазера и тела. В качестве аргумента берется само событие. Из события можно узнать место где оно произошло (e.pos), нормаль столкновения (e.normal), а также получить доступ ко всем свойствам объектов участвовавших в столкновении (e.this.density, e.other.color и проч (в случае с лазером, e.geom. и e.laser.)).

onCollide используется там, где мы не можем заранее вписать код в меню скриптов. Например, объект с таким кодом

```
(event)=>{event.other.color=[1,0,0,1]}
```

Будет окрашивать все объекты, к которым он прикоснулся, в красный цвет.

Приводить еще примеры не буду, так как использование onCollide очень широко, и вы сами уже можете придумать что, где, когда и как.

Урок 10: Функция For

Рассказывая о скриптах, не могу не упомянуть о функции For. Она нужна для того чтобы выполнять одно и тоже действие много раз. Писать ее просто –

```
For(N, (I)=>{})
```

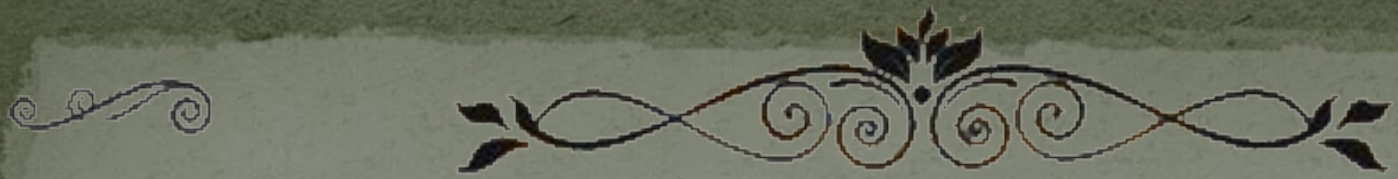
Где N – количество повторов, (I)=>{} выполняемая функция. В качестве I будет использоваться номер повтора (они тоже нумеруются начиная с нуля).

Здесь есть маленькое ограничение на количество повторов – около 50. Но в некоторых исключительных случаях этого может оказаться мало, поэтому был придуман способ обойти это ограничение. Для этого нужно создать свою функцию –

```
scene.my.ExFor := (F, T, P)=>{P < T ? {F(P)} : {};( P + 1) < T ? {F(P + 1)} : {};( P + 2) < T ? {F(P + 2)} : {};( P + 3) < T ? {F(P + 3)} : {};( P + 4) < T ? {F(P + 4)} : {};( P + 5) < T ? {F(P + 5)} : {};(P + 6) < T ? {F(P + 6)} : {};(P + 7) < T ? {F(P + 7)} : {};(P + 8) < T ? {F(P + 8)} : {};(P + 9) < T ? {F(P + 9)} : {};(P + 10) < T ? {Scene.my.ExFor(F, T, (P + 10))} : {}}
```

где F – выполняемая функция, T – количество повторов, P – номер повтора

Можете попытаться проследить, как работает эта функция, и если не запутаетесь, то вы молодец. Скажу только, что она использует рекурсию.



Урок 11: Инфиксные операторы

Хотелось бы рассказать о такой вещи как инфиксный оператор. Инфиксный оператор позволяет нам вместо функции с двумя аргументами писать только аргументы и знак между ними. То есть, по правильному, чтобы сложить два числа, следует писать

```
math.add(x, y)
```

Но вместо этого мы пишем

```
x + y
```

В этом примере знак плюса является инфиксным оператором.

Как прописывать инфикс –

```
infix 2 left _*\*_ => Scene.my.InfixOperationExample
```

Где 2 – количество аргументов для функции, left – порядок чтения функций (left – слева, right – справа), _ - места откуда берутся аргументы.

Но, не следует увлекаться этим делом, ведь инфиксы, как и обычные переменные не сохраняются в сцену…

Урок 12: .phz; .phn Редактирование

Вы наверняка обращали внимание на расширение сейв-файлов для Алгоду – .phz и .phn . Как бы то ни было, их довольно легко редактировать, ведь .phz представляет из себя архив .zip внутри которого лежит контрольная сумма, текстуры и сама сцена в файле .phn

Поэтому .phz легко открывается любым архиватором, а .phn – любым текстовым редактором. Но я бы посоветовал не самый простой блокнот, так как в нем не отображаются переводы строк (Word или Notepad++ и т.д. подойдут).

Внутри .phn файла вы увидите все основные параметры (такие как Название, Автор, настройки сцены и пр.), все объявленные переменные scene.my.* и сами объекты, записанные как scene.add* . Можете поковыряться в нем ради интереса, но особой пользы я не вижу.

Сосем по-другому дело обстоит с другим направлением .phn редактирования. Этот метод я изобрел сам. По крайней мере, я додумался до него сам, и раньше не встречал упоминаний о нем…

Дело вот в чем – иногда появляется необходимость спавна большого количества объектов сложной конфигурации и месторасположения относительно друг друга. В таком случае следует построить данную конфигурацию с помощью инструментов, выделить и скопировать (нажать Ctrl - C), потом создать новый файл в блокноте, и вставить текст (Ctrl - V). Вуаля, теперь у нас есть наша конфигурация, записанная в скрипте!

Теперь нужно удалить ненужные строки вроде FileInfo и комментариев, а также те строки со свойствами scene.add* которые не важны для вас (свойства можно и не удалять, просто тогда будет громоздкий код). Потом надо скопировать код и все, можно вставлять хоть в консоль, хоть в игровое поле, хоть в onCollide. Если вам надо чтобы конфигурация спавнилась в месте контакта, то все строки типа pos := [x, y] надо заменить на pos := e.pos+[x, y]

Урок 13: Добавление объектов

Довольно часто новички спрашивают о том “как сделать, чтобы при столкновении появлялся круг/коробку/…”, поэтому я здесь подробно опишу этот процесс.

Для добавления объектов созданы несколько специальных функций
Scene.AddBox – добавляет прямоугольник
Scene.AddCircle – добавляет круг
Scene.AddPlane – добавляет плоскость
Scene.AddPolygon – добавляет многоугольник
Scene.AddWater – добавляет воду
Scene.AddFixjoint – добавляет крепление
Scene.AddHinge – добавляет ось
Scene.AddLaserPen – добавляет лазер
Scene.AddSpring – добавляет пружину
Scene.AddGroup – группирует объекты

Пример кода для консоли:

```
Scene.addcircle({pos := [0,0]; color := [1,0,0,1]})
```

Это создаст круг красного цвета в центре сцены. Также можно указывать другие параметры, но если не указывать, то им присвоятся дефолтные значения (некоторые из них можно изменить в настройках).

Также можно использовать e.pos и e.normal, например такой код

Scene.addcircle({pos := e.pos; color := [1,0,0,1]})

будет создавать круг в точке столкновения двух объектов. А e.normal часто используется в оружии, чтобы придать созданному снаряду ускорение или создавать объект некоторой точке относительно точки столкновения, например этот код

Scene.addcircle({pos := [0,0] + e.normal*0.5; vel := e.normal*20; radius := 0.25; color := [0,0,0,1]})

Будет создавать круги в отдалении 0.5 метра от точки столкновения в направлении перпендикуляра к касательной точки столкновения, и придавать им ускорение 20 м/с в том же направлении.

Для создания креплений, осей, пружин, лазеров, вам может понадобиться знать такие параметры сталкивающихся объектов, как entityID, geomID, и некоторые другие. Узнать их можно используя New Method (можете прочитать на нашем сайте, или подождать следующую версию книги)

Также следует упомянуть многоугольники – для них надо указывать параметр surfaces, в котором перечислены координаты его вершин. Пример –

Scene.AddPolygon({surfaces:=[[[0,0], [1,1], [0,1]]]})

Урок 14: Разное: Сохранение событий

Вроде бы, все основное я уже рассказал, поэтому теперь расскажу о нескольких интересных методиках. В Thyme есть возможность сохранить событие. Для этого надо в строке onCollide прописать код наподобие такого –

```
(e)=>{scene.my.saveE := {e}}
```

Теперь в переменной saveE сохранено событие, и можно узнать все, что вы узнали бы через просто E (например, scene.my.savee.pos). Довольно интересно сохранить событие, потом в другом объекте сделать пойнтер цвета

```
color = {scene.my.savee.this.color}
```

и изменять цвет объекта, в котором был скрипт сохранения. Если вы так сделаете, то увидите, что цвет объекта с пойнтером тоже изменяется.

Часто говоря, не знаю где можно это использовать, но сам факт, что есть такая возможность, достоин занесения в книгу.

Урок 15: Разное: Удаление через код

Удаление через код - это когда вы можете удалить объект, не устанавливая клавишу уничтожения.

Первый путь – через сохраненное событие
Scene.my.saveE.this.density = 0.

Второй путь – через пойнтер.
Сделайте пойнтер плотности на переменную,
В нужный момент измените переменную на 0 (можно и не самим изменять…).
Также можно использовать onCollide –

```
(e)=>{e.this.density = 0}
```

Что насчет удаления через некоторое время, вот один из способов

```
onCollide = (e)=>{controlleracc=sim.time+3; density={sim.time>controlleracc?
0:2}; onCollide=(e)=>{}}
```

Если вы усвоили материал, то после небольшого анализа поймете, как это работает.

Урок 16: Разное: Программирование клавиш

Есть в Алгоду одна фича – можно запрограммировать какую-нибудь клавишу. Тогда при нажатии клавиши будет выполняться какое-нибудь действие. Это позволяет создавать управляющие системы без создания сложных механизмов изменяющих переменные (как следствие, уменьшаются потери ресурсов). Пример -

```
keys.bind("a",{scene.my.var = 9})
```

(Буквы, опять же, английские…)
Это можно использовать вместо onCollide, так как, onCollide предполагает, что симуляция будет включена, в то время как кибайнд будет работать и без того.
Чтобы распрограммировать клавишу используйте функцию

```
keys.unbind("a",())
```

Правда есть один большой минус, который перекрывает все плюсы – кибайнд не сохраняется, так же как и нестандартные переменные вне раздела scene.my.

Урок 17: Разное: Внутреннее обновление пойнтеров

Иногда при использовании пойнтеров возникает проблема – после сохранения некоторые свойства теряют пойнтеры, или вы можете случайно изменить цвет объекта и тем самым сбить пойнтер.

Чтобы избежать этих досадных недоразумений, заходим в меню скриптов, выбираем свойство, которое вы не собираетесь изменять или которое имеет меньшую вероятность потерять пойнтер, и пишем код вроде такого

```
{length = {Scene.my.var}; false}
```

Где false – исходное значение самого свойства, в моем примере это было свойство opaqueBorders.

Если все равно не помогает, увы, придется использовать onCollide, хотя есть еще один способ… Код примерно такой –

```
scene.my.var = scene.addcircle({radius := 1})
```

Как результат – через scene.my.var мы может изменять свойство созданного объекта…

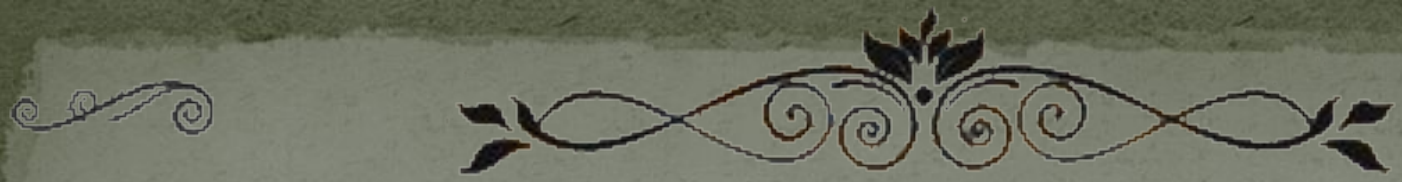
Урок 18: Разное: Обратный пойнтер

В отличие от пойнтера, обратный пойнтер не получает информацию из переменной и передает свойству в котором он прописан, а наоборот, передает в переменную информацию об одном из свойств объекта в котором он прописан. Также, обратный пойнтер пишется почти как внутренний обновитель – выбираете свойство, которое не будете изменять и пишете

{scene.my.getLength = length; true}

Обратный пойнтер можно использовать для слежения за объектом, либо для создания дружественного интерфейса. Например, можно создать обратный пойнтер на то свойство, которое есть в меню настроек объекта и тогда пользователь сможет просто подвигать ползунок, чтобы плавно изменить переменную, а там уже ваши скрипты заработают.

Или, например можно сделать, чтобы при изменении свойства одного объекта, изменялись и у других, это удобно, если эти объекты находятся в труднодоступном месте, а ломать всю конструкцию не хочется.



Урок 19: Разное: Функция Eval

Если хотите стать серьезным скриптером, вам нельзя не знать об этой функции. Кратко говоря, eval выполняет отрывки кода, записанные как текст. Если у вас есть переменная scene.my.txt со значением "scene.my.var=scene.my.var+1"
Вы можете прописать такой код -

```
eval(Scene.my.txt)
```

И все, эффект такой как если бы вы прописали это действие в консоль.
Чтобы понять, зачем его использовать, сравните две функции

```
(errorCode)=>{errorCode == "0" ? {Scene.my.errorCode0} : {errorCode == "1" ? {Scene.my.errorCode1} : {errorCode == "2" ? {Scene.my.errorCode2} : {errorCode == "3" ? {Scene.my.errorCode3} : {errorCode == "4" ? {Scene.my.errorCode4} : {errorCode == "5" ? {Scene.my.errorCode5} : {errorCode == "6" ? {Scene.my.errorCode6} : {errorCode == "7" ? {Scene.my.errorCode7} : {errorCode == "8" ? {Scene.my.errorCode8} : {errorCode == "9" ? {Scene.my.errorCode9} : {Scene.my.errorCodeERROR}}}}}}}}}}
```

```
(errorCode)=>{eval("Scene.my.errorCode" + errorCode)}
```

Здорово, не правда ли?

Урок 20: Разное: Изменение элементов массива

Как известно, нельзя изменить элемент массива просто указав его индекс. Поэтому умным и дотошным скриптерам пришлось придумывать что-нибудь, что могло бы им помочь.

Я привожу здесь готовый код без пояснений, ожидая, что вы уже хорошо разбираетесь в скриптах и можете сами проанализировать этот код и понять, как он работает, и что указывать как аргумент. Можете считать это выпускным экзаменом ☺.

```
Scene.my.ElementChange = (Variable,Element,Size,SetTo) =>
{Scene.my.ElementChangeT (Size, Element, 0, SetTo, [], Variable)};
Scene.my.ElementChangeT = (Size, Element,
CurElement,SetTo,CurArray,OrigArray)=>{Size > 0 ? {CurElement < Size ? {CurElement
== Element ? {Scene.my.ElementChangeT (Size,Element,(CurElement +
1),SetTo,(CurArray ++ [SetTo]),OrigArray)} : {Scene.my.ElementChangeT
(Size,Element,(CurElement + 1),SetTo,(CurArray ++ [OrigArray
CurElement]),OrigArray)};} : {CurArray}} : {print ("Arrays require more than 0
elements")} };
```

Удачи в скриптинге!!!