MID-PROJECT REPORT

https://github.com/pooya79/mid-project

pooya fallah

کلاس Maze:

این کلاس شامل سه متغیر پرایوت است که یکی از آن ها maze وکتوری دو بعدی از shared_ptr است که به متغیرهایی از نوع Node اشاره میکنند این بردار تمام بلاک های میز را در خود نگه میدارد، متغیرهای دیگر کلاس maze، root and goal هستند که حاوی اشاره‌گر به نود شروع و پایان maze هستند.

در پایین ابتدا به توضیح درباره nested class، نود پرداخته شده است سپس به بررسی توابع maze پرداخته شده است.

کلاس Node:

این کلاس که بلاک های maze از این متغیر است شامل دو متغیر پرایوت، mValue که از نوع عدد int است و نوع بلاک را مشخص میکند (0:free, 1=solved path, 2=path, 3=wall, 4=solid wall, 5=startpoint, 6=goal) (منظور از solid wall دیوارهایی است که اطراف maze را فراگرفته اند و wall دیوارهایی که در داخل قرار دارند) و متغیر دیگر isWin از نوع bool است که مشخص میکند نود داده شده نود هدف (پایان maze) است یا نه. برای دسترسی به این دو متغیر getters and setters های مورد نیاز تعریف شده‌اند. این کلاس همچنین دارای سه متغیر پابلیک parent, neighbors and pos است که neighbors لیستی از نودهای اطراف در میز است و pos که از جنس pair است موقعیت نود در درون میز را نشان میدهد.

برای این کلاس یک کانستراکتور با مقادیر دیفالت تعریف شده است تا نیازی به تعریف چند کانستراکتور به عنوان دیفالت کانستراکتور یا کانستراکتورهایی با ورودی کمتر نباشد این تکنیک در بیشتر جاهای کد استفاده شده است که توضیح آن برای کلاس های دیگر داده نشده. همچنین از آنجایی که در این کلاس از smart pointer استفاده شده دیستراکتور تعریف نشده است.

توابع Maze:

constructor:

در کانستراکتور این کلاس دو مقدار size_t دریافت میشود که به ترتیب تعداد ردیف‌ها و ستون‌های maze را بدون در نظر گرفتن برد اطراف maze تعیین میکنند در داخل کانستراکتور با استفاده از دو for loop سایز این وکتور به صورت (n+2*m+2) با درنظر گرفتن برد میز تعیین میشود و در داخل هر کدام از این عناصر

یک پوینتر به Node به صورت shared_Node ایجاد میشود و اگر این نود داخل میز باشد مقدار آن صفر (free) و اگر از بلاک‌های کناری میز باشد مقدار آن 4 (solid wall) در نظر گرفته میشود.

پس از اینکار عنصر 1*1 وکتور میز به عنوان root یعنی شروع میز تعیین میشود و مقدار آن برابر 2 (path) میشود. همچنین عنصر0*1 مقدار 5 را میگرد. پس از این دو srand که برای تولید عدد رندوم در توابع دیگر به آن نیاز میشود مقدار زمان فعلی را دریافت میکند.

در آخر ابتدا تابع createMaze که موقعیت نود شروع را دریافت میکند شروع به ساخت میز میکند و پس از آن با توجه به شکل میز مکان goal در گوشه پایین سمت راست و در برد میز قرار میگیرد، مقدار آن برابر 6 میشود و متغیر isWin در بلاک قبلی آن به صورت true در می‌آید.

:createMaze

همان طور که گفته شد این تابع موقعت شروع میز در وکتور را دریافت میکند و در یک shared_ptr به Node قرار میدهد سپس عددی رندوم از 0 تا 3 ایجاد میکند و وارد while loop میشود که با استفاده از تابع کمکی anyEmptyPath (که با بررسی چهار طرف نود انتخابی اگر صفری وجود داشت true برمیگرداند) شرط گذاری شده در داخل while وارد switch میشود که بر اساس عدد رندم تولید شده جهت حرکت تولید میز را تعیین می‌کند، برای مثال اگر وارد کیس صفر شد ابتدا بررسی میکند که آیا نود با مقدار صفر (free) در آن جهت است اگر نبود عدد رندم دیگری تولید میکند تا زمانی که وارد کیسی شود که نود در آن جهت صفر است. در داخل هر کیس نود در آن جهت و نود فعلی را به عنوان همسایه یکدیگر قرار میدهد سپس مقدار آن را برابر 2 میگذارد سپس از تابع کمکی placeWall استفاده میشود که این تابع مکان نود جدیدی که مقدارش 2 شد را میگیرد و اگر دو همسایه‌ی بعدی آن در جهت‌های مختلف (غیر از جهتی که نود فعلی در تابع createMaze قرار دارد، برای تشخیص این کار بررسی میکند که همسایه اش مقدارش صفر است یا نه) نیز مقدار 2 بود یعنی از مسیرهای میز بود میان آن دو دیوار میگذارد (نود وسطی را مقدارش را برابر 3 میگذارد) این کار از ایجاد loop در میز جلوگیری میکند در آخر هم تابع createMaze را با مکان نود فعلی ایجاد شده صدا میزند.

:solve

این تابع یک مقدار size_t دریافت میکند که نوع الگوریتم حل را مشخص میکند اگر 1 باشد solveDFS را صدا میزند، اگر 2 باشد solveBFS و اگر 3 باشد solveBS را صدا میزند (توضیح این سه در زیر آمده).

پس از آن متغیری از نوع وکتور دو بعدی از int ایجاد میکند به نام mazeData و به سایز وکتور maze که مقادیر نودها را در خود نگه میدارد از این متغیر در بخش GUI استفاده میشود.

solveDFS:

در این تابع پوینتری به نود دریافت میشود که در دفعه اول root یعنی نود شروع کننده است سپس متغیری از جنس bool به نام isWinhere با مقدار false تعریف میشود سپس بررسی میشود که اگر نود فعلی نود هدف بود این متغیر true میشود سپس اگر نود فعلی تابع parent داشت آن از همسایه‌هایش حذف میکند بعد از آن parent همسایه‌هایش را نود فعلی قرار میدهد (هدف از این کار جلوگیری از ایجاد لوپ در الگوریتم است) سپس در یک for میان همسایه‌های نود فعلی تابع را دوباره روی هر همسایه اجرا میکند که اگر نود هدف در آن همسایه یا همسایه‌ای از همسایه‌هایش قرار داشت مقدار isWinhere را برابر true بگذارد سپس در آخر اگر مقدار isWinhere برابر true بود مقدار آن نود را برابر 1 میگذارد و isWinhere را به عنوان خروجی برمیگرداند.

solveBFS:

در این تابع ابتدا بررسی میکند که اگر root با goal یکی بود تابع کمکی drawPath را صدا بزند و از تابع خارج میشود. (که این حالت زمانی اتفاق می‌افتد که میزی 1*1 باشد که با توجه بهGUI این برنامه این حالت ممکن نیست) (تابع کمکی drawPath به صورت بازگشتی مقدار نود فعلی و parentهایش را برابر یک میگذارد.)

پس از آن یک پوینتری از جنس لیست نودها ایجاد میشود به نام tempQueue که در ابتدا همسایه‌های root را در خود قرار میدهد و parent نودهای داخلش را root قرار میدهد سپس وارد حلقه while میشود که تا زمانی که نود هدف پیدا نشود از آن خارج نمیشود در این حلقه یک پوینتر به لیست دیگر ایجاد میشود به نام queue که مقادیر tempQueue به آن منتقل میشود و tempQueue از مقادیرش خالی میشود پس از آن وارد حلقه‌ای میشویم که در عناصر queue گردش میکند اگر نود هدف پیدا نشد ابتدا parent هر عنصر را از لیست همسایه‌هایش حذف میکند و parent همسایه‌هایش را برابر با خود میگذارد سپس همسایه‌هایش را به tempQueue اضافه میکند (برای اینکار از تابع Maze::merge استفاده شده چون merge در std::list اقدام به سورت کردن میکند) در آخر for loop اگر هیچکدام از عناصر queue هدف نبود به اول لوپ while میرود تا tempQueue جدید به آن منتقل شود اگر در یکی از عناصر queue نود هدف پیدا شود (نودی که isWin=true دارد) تابع کمکی drawPath برای آن نود صدا زده میشود و از تابع solveBFS خارج میشود.

solveBS:

عملکرد این تابع شبیه به BFS است با این تفاوت که به جای یک مسیر از دو مسیر که یکی از root و دیگری از goal شروع شده است استفاده شده است. این مسیرها به نوبت یکی در میان در یک for تا زمانی که صف هر دو تمام شود گردش میکنند و در صورتی که در حرکت یکی از این دو مسیری به نودی برسد که قبلا دارای parent باشد برروی آن نود و نود فعلی حلقه تابع drawPath اجرا شده و از تابع solveBS خارج میشود.

show:

میز را برروی ترمینال نمایش میدهد که برای این برنامه ناکارآمد است.

GUI:

ایده کلی این بازی به این صورت است که با انتخاب کاراکتر و اژدها موردنظر بازیکن باید از میز گذشته و به اژدهایی که در حال خواب یا استراحت است برسد اگر نمیتواند با زدن گزینه character go میتواند کاری کند تا کاراکتر به طور خودکار مسیر را براساس الگوریتم انتخاب شده طی کند همچنین با زدن awake dragon عکس اژدها تغییر کرده و مسیر درست میز با توجه به نوع اژدها مشخص میشود.

در گزارش نحوه کار GUI ابتدا توضیحی کلی درمورد فایل‌های main.cpp، res.qrc و کلاس game داده میشود و سپس به نحوه ایجاد هر یک صفحات برنامه (start menu, option menu, game menu) پرداخته میشود.

فایل main.cpp:

در فایل main ابتدا یک متغیر از نوع QApplication ایجاد شده که برای ایجاد GUI در همه برنامه‌های qt معرفی میشود که در آخر تابع main با return کردن تابع exec آن برنامه اجرا میشود. از این متغیر برای ایجاد آیکون برنامه که از عکسی داخل فایل res انتخاب شده و همچنین مشخص کردن سایز screen، کاربر استفاده میشود. سایز اسکرین برنامه به صورت دیفالت 800*1200 است که تمام itemهای برنامه بر اساس آن تنظیم شده‌اند (این برنامه interactive نیست و برای اجرای مناسب باید نسبت سایز 3 به 2 باشد) اگر اسکرین کاربر از 1200 طول یا 800 عرض کمتر باشد برنامه به مقدار 0.75 اسکیل میشود یعنی با سایز 600*900 اجرا میشود. در آخر متغیر game از نوع اشاره‌گر به کلاس Game که نسبت سایز به آن داده‌شده است ایجاد میشود و تابع show آن و سپس displayStartMenu برای نمایش start menu اجرا میشود.

فایل res.qrc:

این فایل که از فایل‌های ایجاد شده توسط qt-creator است مسئول نگه داشتند تمام عکس‌ها و موسیقی‌های درون بازی است با وجود این فایل در صورت پک کردن برنامه تمام عکس‌ها و موسیقی‌ها داخل فایل exe. قرار میگیرند و دیگر نیازی به داشتن پوشه‌ای حاوی عکس‌ها و موسیقی‌ها در کنار آن نیست. (همچنین با اینکار امکان crossplatform برای دسترسی به فایل‌ها ایجاد میشود)

کلاس game:

این کلاس که ارث برده از کلاس QGraphicsView مسئول نمایش و کنترل تمام آیتم‌های بازی است که به scene که یکی از متغیرهای پابلیک همین کلاس از نوع اشاره‌گر به QGrahpicsScene است اضافه شده‌اند.

دیگر متغیرهای این کلاس به صورت پرایوت تعریف شده‌اند که شامل sizeRatio که اندازه صفحه و آیتم‌ها را براساس آن اسکیل میکند و همچنین متغیرهای دیگر است که در پایین به ترتیب توضیح داده میشوند. در این کلاس Q_OBJECT نوشته شده است که با نوشتن آن در کلاس‌ها امکان استفاده از slots/signals که از قابلیت‌های qt در ایجاد GUI است را میدهد.

در کانستراکتور این کلاس تنظیمات اسکرین بازی شامل نبود scrollbar، سایز صفحه (براساس sizeRatio) و background منو شروع انجام شده است همچنین مقداردهی‌های اولیه متغیرهایی مانند متغیری که مربوط به منو تنظیمات، پلی لیست و موزیک‌های مربوط به موزیک پشت صحنه منو بازی و موزیک هنگام پیروزی انجام شده است.

در دیستراکتور این کلاس تنها scene و متغیرهای مربوط به موسیقی‌ها حذف شده‌اند.

(علت اینکه در این کلاس و دیگر کلاس‌های برنامه با وجود اینکه بیشتر متغیرها با new بوجود آمده‌اند و دیستراکتور لازم وجود ندارد این است که از قابلیت‌های qt این است که تمام آیتم ها حاوی parent بوده که با حذف parent تمام بچه‌های آن نیز delete میشوند بنابراین با حذف بالاترین parent که اینجا scene است تمام آیتم‌های داخل آن حذف میشوند)

Start Menu:

این صفحه شامل background و باکسی با گزینه‌های Start برای ورود به option menu و Quit برای خارج شدن از برنامه است.

background این صفحه در کانستراکتور game ایجاد میشود.

باکس موجود با صدا زدن تابع displayStartMenu، کلاس Game ایجاد میشود اینکار تنها یکبار و آن‌هم همانطور که قبلا گفته‌شد درون فایل main اتفاق می‌افتد.

ابتدا border باکس تعریف میشود که یک اشاره‌گر از نوع QGraphicsPixmapItem (این متغیر میتواند تصاویر را نمایش دهد) با عکسی از بوردرهای تعیین شده است سایز و مکان این آیتم براساس sizeRatio تعیین میشود و به scene اضافه میشود.

برای ایجاد دکمه‌ها کلاس Button تعریف شده است این کلاس که ارث برده از QGraphicsPixmapItme حاوی محل عکس‌های مربوط به Button, hoverButton, clickedButton است با گرفتن اسم Button عکس‌های مربوطه را که از قبل در فایل res قرار داده شده اند را در متغیرهایش قرار میدهد و با هاور کردن، کلیک کردن و خارج شدن از دکمه عکس مورد نظر را نمایش میدهد همچنین با کلیک برروی آن سیگنالی را منتشر میکند. (slots/signals از قابلیت‌های qt)

دو دکمه start و quit از نوع اشاره‌گر به Button تعریف میشوند و parent این دو بردری که قبلا گفته شد قرار داده میشود و مکان آن‌ها نسبت به بردر تعیین میشود.

در آخر دکمه start به اسلات startAndNext که با کلید دکمه‌های start و next (در منو بازی) صدا زده میشود connect (از قابلیت‌های qt برای ایجاد ارتباط میان signals and slots) میشود این اسلات وظیفه نمایش منو تنظیمات را دارد. و دکمه quit به close که تابعی برای خروج از برنامه است وصل میشود.

Option Menu:

در این صفحه کاربر تعداد ردیف‌ها، تعداد ستون‌ها، نوع الگوریتم حل میز، نوع کاراکتر و نوع اژدها را تعیین میکند. با انتخاب کاراکترها و اژدهاهای مختلف عکس‌ها و موزیک‌های متفاوتی اجرا میشود.

برای ایجاد منو کلاسی به نام Option تعریف شده است این کلاس که وظیفه کنترل و نمایش تمام اجزای منو تنظیمات را دارد از کلاس QGraphicsPixmapItem ارث برده شده است. این کلاس شامل متغیری به نام data از نوع آرایه‌ای از اعداد صحیح به سایز 5 است که داده‌های مربوط به گزینه‌های انتخاب شده را نگه میدارد و با تابع getData به برنامه میدهد.

در کلاس Game متغیری از نوع اشاره‌گر به کلاس Option وجود دارد که ابتدا در کانستراکتور این کلاس new میشود و کانستراکتور آن صدا زده میشود و با ورود به تابع startAndNext تابع Option::setUp صدا زده میشود این تابع که از چهار تابع کمکی تشکیل شده است وظیفه رسم اسلایدرها و دکمه‌های رادیویی را دارد. در رسم اسلایدرها (برای تعیین ابعاد میز) تابع کمکی drawSliders در Option معرفی شده است، در آن برای رسم QSlider ها از QGraphicsProxyWidget استفاده شده که اجازه رسم اسلایدرها را در آیتم‌های QGraghphics میدهد، مقدار مینیمم و ماکزیمم اسلایدرها برابر 2 و 100 گذاشته شده و هرکدام با اسلاتی مربوط به خود connect شده که در صورت تغییر در اسلایدر تکستی که در کنار هر اسلایدر است (این تکست در کانستراکتور Option تعریف شده) و داده مربوطه در متغیر data (دو عضو اول مربوط به تعداد ردیف‌ها و ستون‌ها است) را تغییر دهد. همچنین برای تغییر ظاهر اسلایدر از setStyleSheet که کارایی شبیه به CSS دارد استفاده شده.

برای تعریف گزینه‌های مربوط به انتخاب نوع الگوریتم تابع کمکی drawSolveAlg تعریف شده که در آن از متغیرهایی از جنس RadioButton استفاده شده است که کارکرد آن‌ها مانند Button است با این تفاوت که یک متغیر checked برای تعیین اینکه روی آن‌ها کلیک شده و یک متغیر دیگر به نام id قرار دارد که برای مثال در تابع Option::drawSolveAlg سه تا از این متغیرها یکی با id 1 که نشان دهنده DFS است، دیگری با id 2 که نشان‌دهنده BFS است و در آخر id 3 که نشان‌دهنده BS است قرار دارد. این سه دکمه رادیویی به اسلات Option::algChanged که تغییرات لازم در آن‌ها (تغییر در checked و تصاویر آن‌ها) و همچنین تغییر در عنصر سوم آرایه data را انجام میدهد. دو تابع کمکی برای رسم radio buttonهای نوع کاراکتر و اژدها نیز مانند drawSolveAlg عمل میکنند تنها با این تفاوت که از چهار radio button تشکیل شده‌اند.

پس از اجرای Option::setUp در Game::startAndNext دکمه Begin تعریف میشود، به تابع Game::startGame، connect میشود (این تابع وظیفه اجرای منو بازی را دارد) و موقعیت آن براساس sizeRatio در منو تنظیمات تنظیم میشود. در آخرهم scene->clear() اجرا شده که تمام آیتم‌های قبلی از منو استارت را حذف کرده (delete) و سپس با scene->addItem، آیتم optionMenu را اضافه میکند.

Game Menu:

همانطور که گفته شده دکمه Begin با تابع startGame، connect میشود که با کلیک برروی آن این تابع اجرا میشود. در این تابع ابتدا داده‌های موجود در optionMenu (آرایه 5 تایی از int که نشان‌دهنده ابعاد،

نوع‌الگوریتم و.... است) به متغیری در کلاس Game داده می‌شود سپس صحنه با clear خالی میشود، background آن عوض میشود و موزیک پس‌زمینه اجرا میشود (برای تعریف این موزیک از آنجایی که موزیک پس‌زمینه باید پس از اتمام دوباره اجرا شود یک playlist هم همراه آن تعریف شده که به صورت loop آهنگ انتخابی را اجرا میکند این تعاریف در کانستراکتور Game انجام شده است) (موسیقی‌های اجرا شده در تابع startAndNext یعنی در زمان ورود به منو تنظیمات متوقف میشوند.)

سپس شروع به کشیدن برد بازی یا همان میز با استفاده از کلاس Board میکند. در کاسنتراکتور این کلاس ابتدا sizeRatio و داده‌ها (اطلاعات گرفته شده از option menu) را مقداردهی میکند سپس تابع کمکی setUp را فرا میخواند در این تابع ابتدا اقدام به کشیدن بلاک‌های میز میکند برای اینکار ابتدا براساس sizeRatio سایز بلوک‌های و موقعیت اولین بلوک را طوری تعریف میکند تا میز در وسط مکانی که برایش مقرر شده جای بگیرد پس از اینها متغیر maze از کلاس Maze با توجه به اندازه‌های گرفته شده از منو تنظیمات ساخته میشود و با صدا زدن Maze::solve با الگوریتم مورد نظر داده‌های مربوط به آن در mazeData ذخیره میشود (این آرایه میز حل شده‌ای است که از آن برای رسم میز حل نشده و حل شده استفاده میشود) سپس تابع کمکی drawMaze صدا زده شده که شروع به کشیدن به بلاک‌های میز براساس شماره آن‌ها از آیتم هایی از نوع QGraphicsPixmapItem میکند که تمام این آیتم‌ها را درون وکتوری دوبعدی ذخیره میکند تا در هنگام حل کردن به میز به آن‌ها دسترسی داشته و تغییرات لازم را انجام دهد همچنین در این حین مکان هدف را در متغیرgoalPos از نوع pair ذخیره میکند. پس از رسم بلاک‌ها در ادامه Board::setUp مکان کاراکتر را در charPos از جنس pair ذخیره کرده و character را از جنس Character تعریف میشود.

کلاس کاراکتر کلاسی است که ظاهر کاراکتر را بر اساس data گرفته شده از منو تنظیمات (این دیتا از کلاس Board به Character منتقل میشود)، موقعیت آن، سایز کاراکتر که هم اندازه سایز بلاک‌های میز است تعیین و کنترل میکند همچنین این کلاس حرکت کاراکتر را با کلیدهای اشاره کنترل میکند برای حرکت کاراکتر این کلاس متغیر mazeData که وکتور دوبعدی حاوی اطلاعات بلاک‌ها است را دریافت میکند تا کاراکتر برروی path حرکت کند همچنین اگر کاراکتر به نقطه هدف رسید که در بازی به صورت یک دروازه مشخص شده سیگنالی به اسم win را منتشر میکند که در ادامه از نحوه کار آن توضیح داده میشود. از کارهای جدید این کلاس که در نسخه جدید به آن اضافه شده امکان حرکت کاراکتر بصورت خودکار با توجه با الگوریتم انتخاب شده‌اشت برای اینکار توابعی مانند توابع حل میز که در کلاس maze استفاده شد تعریف شده است به طوریکه در حل

BFS و DFS کاراکتر با توجه به الگوریتم حرکت کرده و از خود ردپایی به جا میگذارد و در حل BS چون حرکت از دو جهت صورت میگیرد علاوه بر کاراکتر از مکان goal متغیری به اسم babyDragon نیز شروع به حرکت کرده که در صورت رسیدن این دو به هم کاراکتری مسیر ردپای بچه اژدها را دنبال کرده و به هدف میرسد.

در ادامه تابع Board::setUp پس از تعریف کاراکتر موقعیت آن برروی برد تنظیم میشود و سیگنال win آن به یک lambda function متصل میشود که سیگنال Board::win را منتشر کرده و با اجرای character->clearFocus اجازه حرکت را از کاراکتر میگیرد.

در آخر Board::setUp با اجرای دو خط کد کاراکتر را focus برنامه قرار میدهد این کار باعث میشود که کلیدهای keyboard برروی کاراکتر اجرا شوند.

حال که تعریف board به اتمام میرسد به تابع Game::startGame بازمیگردیم در این تابع پس از تعریف board سیگنال win آن به اسلات Game::win متصل میشود این اسلات موسیقی background را متوقف کرده موزیک پیروزی را پخش میکند و آیتمی که نمایشدهنده پیروزی است را در side menu (نحوه تعریف پایین تر) نمایش میدهد.

پس از رسم board تابع Game::startGame شروع به کشید side menu میکند نحوه کشیدن این منو مانند منو صفحه استارت برنامه است در این منو Button 4 وجود دارد Next که به startAndNext متصل است، Awake Dragon که به اسلات awakeDragon (این اسلات پایین آخر گزارش توضیح داده شده است) متصل است، QUIT که به Game::close وصل است و باعث خارج شدن از برنامه میشود و characterGo که به تابع characterGo در Game وصل است (این تابع یک thread ایجاد کرده و حرکت کردن کاراکتر با توجه به الگوریتم را کنترل میکند) همچنین در مکان نمایش دکمه characterGo یک اسلایدر ایجاد میشود که با زدن این دکمه ایجاد شده و امکان کنترل سرعت حرکت کاراکتر را به کاربر میدهد. در پایین side menu فضای بازی وجود دارد که پیام مربوط به پیروزی یا شکست نمایش داده میشود.

پس از side menu، متغیر dragon از جنس Dragon تعریف میشود و پس از تعیین موقعیت به scene اضافه میشود. کلاس Dragon که مسئول تنظیم اندازه، عکس اژدها و همچنین صدای اژدها است sizeRatio و نوع اژدها را از Board میگیرد. (نوع اژدها مانند کاراکتر به صورت عدد صحیح در data قرار دارد)

در آخر متغیر optionMenu که از جنس Option بود دوباره new میشود چرا که با زدن دکمه begin و وارد شدن به منو بازی scene->clear() اجرا شده که باعث delete آن میشود.

زمانی که بازیکن در بازی گزینه awake dragon را بزند اسلات Game::awakeDragon اجرا میشود این تابع ابتدا موسیقی background را متوقف کرده سپس dragon->fireDragon را اجرا میکند که در کلاس Dragon تعریف شده و باعث تغییر عکس اژدها و ایجاد صدای آتش اژدها میشود سپس board->solve اجرا میشود که در Board قرار داشته و عکس مربوط به بلاک‌هایی که مسیر درست میز است را عوض میکند در آخر پیغام مربوط به شکست را در side menu نمایش میدهد.